User's Manual
and Programmer's Guide

Modeling Evaluation and Research of Lightwave Networks

version

1.0

U

UNIX

User’s Manual
and Programmer’s Guide

Modeling Evaluation and Research of Lightwave Networks

version

1.0

Authors
Frederic Mouveaux
Francois Lapeyrere
Nada Golmie

Programming
Francois Lapeyrere
Frederic Mouveaux
Guillaume Lhoste

National Institute of Standards and Technology
100 Bureau drive, Stop 8920
Gaithersburg, MD 20899-8920

World Wide Web
w3.antd.nist.gov

Copyright

Modeling Evaluation and Research of Lightwave Networks (MERLiN)
National Institute of Standards and Technology (NIST)

Trademarks

Contents

Introduction

Modeling Evaluation and Research of Lightwave Network (MERLiN) is a Wavelength Division Multiplexing (WDM) network design and modeling environment that allows the development and evaluation of algorithms for wavelength assignment, routing, dynamic reconfiguration and restoration mechanisms. MERLiN can be extended to implement network management procedures and Quality of Service (QoS) models for WDM networks protocols.

About this Manual

This document intends to provides a detailed overview of MERLiN architecture and its major building blocks. It is intended for the network planner who wants to run a simulation of the protocol, the researcher who wants to know the basics about the simulator and how to implement and run his/her own algorithm. It is also for the application programmer who plans to integrate a new component like a plug-in. However, the experienced programmer familiar with the concepts of MERLiN may prefer to use the MERLiN Reference Manual, located in the *www* directory of the archive to search for a specific function, and will refer to this manual only when trying out new features. This document is also available online at *http://grabber.ncsl.nist.gov/~merlin/www.*

In this manual, we assume that the reader is familiar with his/her operating system and its conventions. Most of the sections describe how to use the simulator and suppose that the software has already been installed. The reader is referred to the *Getting Started* chapter in order to install the simulator.

MERLiN user

The first part of this document describes how to use the tool. It contains the following sections:

What is MERLiN ? contains the basic concepts about the MERLiN architecture, the goal of the project and future extensions.

Getting Started describes the system requirements, the installation instructions

The Configuration File gives the file format and a description of the components in order for the user to build his/her own configuration file.

The Graphical User Interface gives an overview of the GUI.

MERLiN Programmer

The second part that is destined for experienced programmer consists of the following chapters:

The Architecture Overview gives the full design of the simulator including the core, the plug-in and the algorithm interface.

The Component Hierarchy presents all the objects and the relations between them, as they are implemented in MERLiN.

The MERLiN Grammar explains the basic concepts about how to read a grammar, and a full description of the MERLiN grammar.

Extending MERLiN explains the use of the template file in order to create new plug-ins and algorithms to be integrated in MERLiN.

Conventions

In order to improve readability and to keep consistent we ???? the following conventions and notations throughout this manual:

Control+n	Holding down Control and pressing the lowercase letter *n.*
italic characters in a paragraph	Emphasis of the corresponding word in the sentence.
centered italic characters	Refer to a command that needs to be entered at the shell prompt.
Arial Bold characters at the beginning	Reference to a title of a paragraph in the document.

Part 1: User's Manual

CHAPTER 1

What Is MERLiN?

Management And Control Of WDM Networks

Optical networking has experienced considerable progress in recent years. For example, wavelength division multiplex (WDM) transport systems with tera-bit/sec capacity on a single strain of fiber is commercially available now. However, a number of technical challenges still remain to be addressed before optical networks become the effective basis for a robust, agile, super-fast, and scalable next generation information infrastructure.

One such challenge is the development of standardized architectures and protocols for network and optical layers, appropriate at very high speeds, and for different environments (e.g., local access/metro and long haul). While experimentation on DARPA sponsored testbed will reveal useful field information about these issues, a simulation tool which can provide rapid and accurate analysis is now essential for timely resolution of the challenges of the integration of networking and optical layers. Building such a network evaluation tool with a high degree of link and optical layer detail (which is required at high speeds such as in excess of OC-48, for DWDM, and for low BER) is now feasible using physical device/component modeling and simulation tools that are under development or commercially available.

Second, a variety of optical networks (based on SONET, WDM, and perhaps TDM) will coexist with legacy networks in the next generation infrastructure. Accordingly, NGI network engineering tools must incorporate existing and emerging optical networking technologies, which have many design features different from those of conventional networks. For example, the distinction between virtual and physical topology designs is made in WDM networks. Network engineering issues also include design at different time scales: from capacity provisioning to handle traffic growth done on a scale of months, to split-second dynamic reconfiguration, with attendant packet/connection management, as an efficient response to fast traffic changes and mission critical applications. Additionally, multiple services with QoS differentiation must be optimally mapped to light-paths, and parameters of protection/restoration functions at optical and electronic layers must be engineered to avoid destructive interference. Furthermore, the accuracy of such tools is enhanced by incorporating optical network management and control functions either via models or by interfacing with actual management and control objects.

The High Speed Network Technologies group in the Advanced Networking Technologies proposes to develop an optical network modeling and planning tool (MERLiN), which is:

1) capable of rapid evaluation of protocols and architectures, with respect to their characteristics in different environments; and 2) also a network engineering tool that can inter-work with higher layer models and device level simulators.

MERLiN has a rich library of algorithms for optimum network design, dynamic reconfiguration, restoration, path protection, wavelength assignment, and performance analysis. It is designed to permit varying levels of abstraction of the optical layer. Specifically, a higher layer network simulator can have models of links and light paths parameterized with features determined/queried by a detailed optical layer simulator; or the latter can be a pluggable module into the overall simulation of all layers. The core of the MERLiN simulation tool provides the capability to compose a physical topology, optimize virtual topology design for given traffic and end-to-end requirements, and overlay Internet Packets through the optical network. In addition to a rich library of topologies, algorithms, and protocols, the tool provides for the creation and evaluation of user-defined algorithms and protocols.

Overview

Modeling Evaluation and Research of Lightwave Network (MERLiN) is a Wavelength Division Multiplexing (WDM) network design and modeling environment that is currently under development at the National Institute of Standards and Technology. The main purpose for this tool is to allow the development and evaluation of algorithms for wavelength assignment, routing, dynamic reconfiguration and restoration mechanisms without the expense of building a real network. There are two major uses for MERLiN: as a tool for WDM network planning and as a tool for WDM protocol performance analysis.

As a planning tool, a network planner can run the simulator with various network configurations, traffic loads to obtain statistics such as wavelength utilization, throughput rates, blocking probabilities. It could be used to answer questions such as: what is shortest path between two nodes, how many wavelengths are utilized on a path, will adding a new connection cause congestion. Statistics and routes are displayed directly on the screen or could be logged in a data file for further processing.

As a protocol analysis tool, a researcher or protocol designer could study the total system effect of particular prototcol. For example, one could investigate the effectiveness of various reconfiguration algorithms for WDM networks and address such issues as: mechanisms for dynamic provisioning, protocol overhead, throughput. MERLiN can be easily extended to implement additional optical network control and management procedures such as performance monitoring, quality of optical service provisioning, and fault detection and recovery.

The architecture of MERLiN consists of three major building blocks, namely, the core, the plug-ins and the algorithms’ library. Each block along with its components is a separate program that handles a specific aspect of the simulation process. This architecture provides a modular plug-and-play platform that lets users add new components such as algorithms and plug-ins and run them on different machines. The main reason for this distributed design is to facilitate the utilization of several computational resources (cluster of workstations, supercomputer) and distributed data bases as needed. It addresses both scalability issues related to the management of thousands of nodes in large networks as well as computational complexities associated with wavelength assignment and routing algorithms. Another advantage of this design is that it allows for language/system independent plug-ins and algorithms. A programmer will be able to access MERLiN without reading the core source code, since a standard communication interface is provided with the simulator. Figure 1 describes the architecture of MERLiN and the communication between the different blocks.

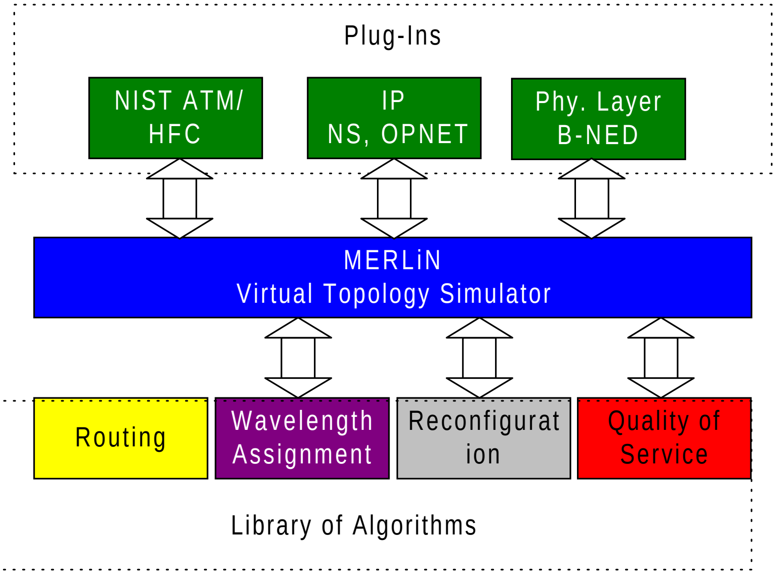
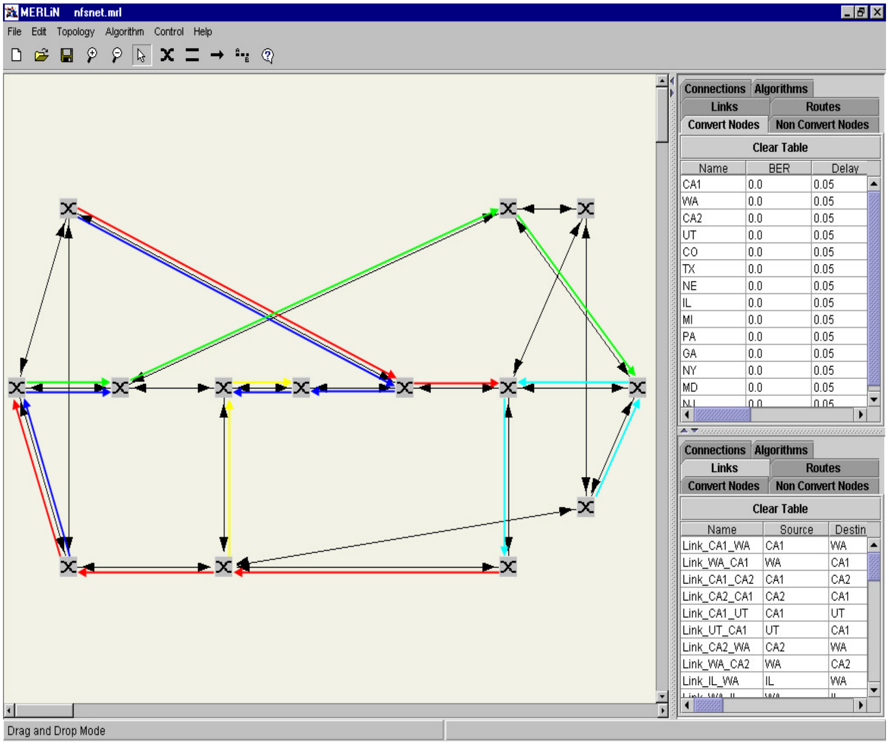


FIGURE 1. MERLiN Global Architecture

MERLiN is based on a server/client architecture. The core is the main entity that waits for user requests, analyzes the input parameters and dispatches the data to the algorithms. It is implemented on the server's side and acts as a daemon that is run in the background. Its main task is to build the network vitual topology from a configuration specified by the user that includes a physical network topology (links, crossconnects, add/drop multiplexers) and a connection request matrix (source and destination pairs) and dispatches it to the algorithms. It then collects the results from the algorithms and send them back to the user via the plug-in interface (in text form or GUI). There are about a dozen routing and wavelength assignment algorithms currently in the library including: shortest path, and K-shortest path for routing, first fit, least used, most used and Tabu for wavelength assignment. The plug-in interface provides a flexible user's access to MERLiN. There are several types of plug-ins such as simple text file, or interface to other simulator packages (physical layer, packet level simulators). MERLiN's Graphical User Interface (GUI) is also considered a plug-in.



Name	BER	Delay
CA1	0.0	0.05
WA	0.0	0.05
CA2	0.0	0.05
UT	0.0	0.05
CO	0.0	0.05
TX	0.0	0.05
NE	0.0	0.05
IL	0.0	0.05
MI	0.0	0.05
PA	0.0	0.05
GA	0.0	0.05
NY	0.0	0.05
MD	0.0	0.05
NJ	0.0	0.05

Name	Source	Destin
Link_CA1_WA	CA1	WA
Link_WA_CA1	WA	CA1
Link_CA1_CA2	CA1	CA2
Link_CA2_CA1	CA2	CA1
Link_CA1_UT	CA1	UT
Link_UT_CA1	UT	CA1
Link_CA2_WA	CA2	WA
Link_WA_CA2	WA	CA2
Link_IL_WA	IL	WA

FIGURE 2. MERLiN GUI

Since measurements of network elements and optical signal characteristics are crucial for developing effective management schemes for WDM networks, a plug-in is currently being developed to interface with several device and component level simulators commercially available such as ARTIS/OPTSIM and Virtual Photonics/ BNeD. These interface modules allow the users of MERLiN to include a high degree of link and optical layer performance characteristics and accuracy in their optical component models (crossconnects, converters, amplifiers, regenerators, fiber links) such as BER, crosstalk, jitter, and power level.

Core

MERLiN is based on a client/server architecture where the core is implemented on the server's side. It is acting as a daemon that is run in the background, and its main task is to prepare and dispatch the data for the simulator plug-ins and algorithms. The core collects the results of the simulation and sends them back to the user via the plug-in interface.

Plug-ins

A plug-in allows a user to access MERLiN and the algorithm database. The role of a plug-in is to send user's requests to the core over the network interface, and collect the results once the simulation terminates.

The information exchanged between the plug-in and the core is in the form of a text script file describing the network component and topology including a list of connection requests between pairs of source and destination nodes, and parameters for the routing, reconfiguration algorithms to run.

The file has to obey a number of syntax rules syntax in order for MERLiN to understand the requests and the different objects describing this file can be generated by different programs such as the *Graphical User Interface* or the output of another simulator after it has undergone the necessary translation. The *Graphical User Interface* help the user to build his/her own topology and requests by simple mouse clicks. It will automatically generate the file and send it to MERLiN when requested. The plug-in interface allows the user to use the results or file format from other simulators and interface them to MERLiN. Through dedicated plug-ins it is possible to get information needed from a physical layer simulator or higher layer simulator (like the NIST ATM/HFC simulator) and translate the information into the format recognized by the core. The example given in the Figure 3 on page 21 is the full process description of the data exchange between the different simulator entities in order to make a physical layer simulator (OptSim) communicate with MERLiN and routes and allocate wavelength given a list of connection requests.

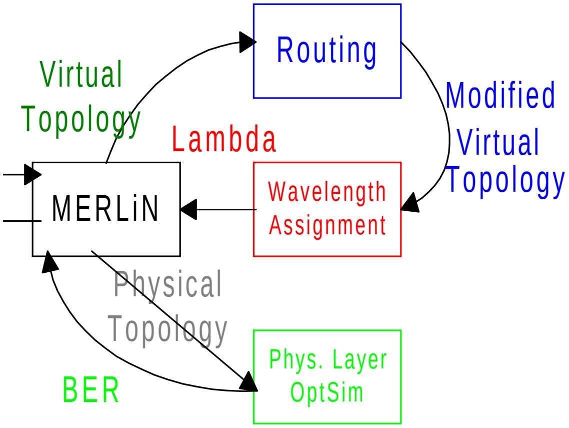


FIGURE 3. Data Exchange in MERLiN

The script file generated is sent to the communication API of MERLiN (based on the UNIX socket interface) and is transported to the core by the physical network cable. After building its internal data structure (i.e. the network virtual topology) the core calls the algorithm and sends its newly created data structure. The *Routing* algorithm takes the virtual topology and finds routes to the connections requested. Then the *Wavelength Assignment* algorithm tries to find an available wavelength for the pre-determined route. The algorithms return to MERLiN the results of the simulation by modifying the input data given to them and adding routes and wavelengths to the traffic request. The results can be displayed in the *Graphical User Interface* as shown in Figure 4 on page 22.

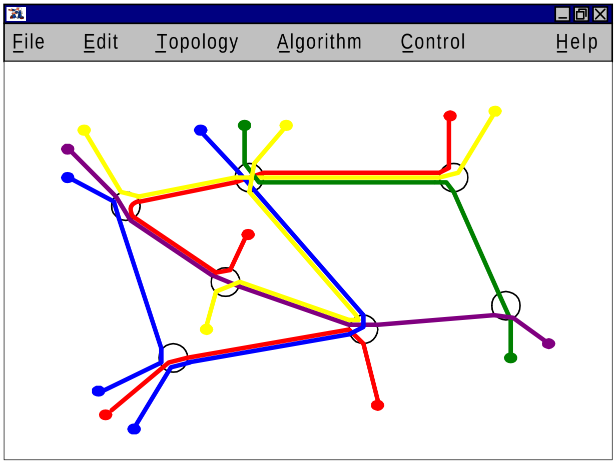


FIGURE 4. **Example of results in the Graphical User Interface**

Algorithms

The library of algorithms contains several classes depending on the type of operation the simulator is asked to perform. Four categories have been designed so far and they include *Routing*, *Wavelength Assignment*, *Reconfiguration*, and *Restoration*. Each category contains a list of algorithms. It is important to understand that the algorithm database is tightly associated with MERLiN's internal representation of the network topology.

Future Extensions

CHAPTER 2

Getting Started

MERLiN runs on UNIX systems and has been tested under Silicon Graphics/Irix, Sun/Solaris, and PC/FreeBSD environments. There are no MS-DOS or Windows 95/98 versions available at this time. Some libraries are required in order to build the executable code associated with the simulator. If you any problems due to missing library files, the user is advised to his/her system administrator. In this chapter, the reader will learn the basic features of MERLiN including installation tips in order to get familiar with the MERLiN tool and get started.

MERLiN Installation gives detailed instruction on how to install the software archive.

Directory Structure is an overview of the archive directory tree and is useful to understand the organization of the different modules of MERLiN.

Running A Simulation provides an example of how to use the text plug-in in order to ask MERLiN to launch some of algorithms located in the database.

Analyzing The Results describes the interpretation of simulation results of a simulation returned by MERLiN.

MERLiN Installation

In order to install the simulator on a given machine, you first have to decompress the archive. This can be done using a combination of two utilities *tar* and *gzip*. If you encounter any errors in the installation process of the software, please refer to the *Frequently Asked Questions* section and check with your system administrator that both the utilities and libraries are correctly installed on your machine. In the directory where the compressed archive is located, type the following command in your shell, where *X* and *Y* are the version number and the revision number of the software package respectively:

```
gzip -cd merlin++_vX.Y.tar.gz | tar xf -
```

At this point, you should have the directory tree structure. This structure will be presented in details later in the chapter. presented in the next point of this chapter. In order to build the executable codes needed to run a simulation, you have to go in the source directory. Please enter at the shell prompt the following command:

```
cd merlin++_vX.Y/src
```

Then you need to use the *configure* script in order for MERLiN to detect the components installed on your system and needed in the compilation process. The configure script allows the user to choose several option such as the destination directory for the binaries, or the debug mode for developers only. For the novice user, the use of this script is straightforward. More details concerning the options of the script can be found in the section *Configure Script* in the Appendix. For a quick installation, no options need to be selected any options. At the prompt in the current directory type :

```
./configure
```

If no errors occur, you can compile the entire application by entering the following command at the prompt:

```
make ; make install
```

This will compile all the modules of MERLiN and install the software in the installation directories. In order for the core of MERLiN to find the algorithms on your computer, you must specify an environment variable named *MERLINPATH*. This variable must point to your installation root directory. This can be done in several ways depending on your shell. We recommend to set this variable in your shell resource file (.tcshrc, .bashrc, etc.). Here is a way to do it under two of the most commonly used shells. Just edit your resource file and add the corresponding line at the end:

under TCSH *setenv MERLINPATH /home/my_local_directory/merlin++*

under BASH *declare -x MERLINPATH="/home/my_local_directory/merlin++"*

After restarting a new shell, you are now ready to use MERLiN.

Directory Structure

The MERLiN archive is organized in several parts in order to allow for easy retrieval and efficient storage of the information. Some of the directories are created when the archive is decompressed, and others are created by the installation process. After installing the simulator in a directory, the archive contains the following main tree of directories:

merlin++	Root directory
	Created by the installation process
algorithm	Binary code of the algorithms
routing	Routing algorithm class
wavelength	Wavelength algorithm class
reconfiguration	Reconfiguration algorithm class
qos	Quality of Service algorithm class
bin	Binary code of the daemon
lib	Merlin development library
plug-in	Plug-ins
	Created when decompressing the archive
www	Reference manual of the classes (HTML format)
example	Some topologies, and examples of MERLiN files
src	Sources of Merlin, Plug-ins and algorithms
algorithm	Sources of all the algorithms implemented
routing	Routing algorithm class
wavelength	Wavelength algorithm class
reconfiguration	Reconfiguration algorithm class
qos	Quality of Service algorithm class
merlin	Sources of MERLiN core

components	Components used in the topologies
math	Standard types library
merlin	Core sources
network	Network API
parser	Input file parser
tool	Development tool library
plug-in	Sources of the Plug-ins
template	Templates of the communication API and Makefiles

Running A Simulation

After completing the installation process, you now ready to run a simulation.

Launching the daemon

The daemon is the core of MERLiN. In order to use it, just launch the executable file that is located in the *bin* directory of this archive.

./merlin

It is important to set correctly the MERLINPATH variable on the machine where you are running the daemon. Please refer to the previous section concerning the installation of the simulator in order to set the variable. The daemon waits until a plug-in connects to it, and then based on the requests specified in the plug-in, launch the appropriate algorithm. If you only want to check the version of the simulator that you are running, launch the daemon with the -V option. Try the following command in your shell:

./merlin -V

This will display a message indicating the version and revision number of the software, and the operating system on which it has been compiled. After displaying the message, the core of MERLiN stops. If you want to run a simulation, you have to launch the daemon again without the -V option.

Using MERLiN plug-ins.

The plug-ins are located in the *plug-in* directory of the archive. They all require a basic set of parameters and for some plug-ins additional parameters are required. The port number requested by the application is the port numbered *2507*. This port must be available in order for a normal operation. The set of parameters for a standard plug-in is:

Plug-in name	Daemon’s machine	Port number	[Extra parameters]

FIGURE 5. **Plug-in parameters**

Plug-in Examples

The plug-in KILL is a basic program that allows you to end remotely the daemon when this one is running on a distant machine. It only takes the basic set of parameters and can be launched as follows:

```
kill localhost 2507
```

Another plug-in is the MERLiN text file application that takes a configuration file, sends it to the daemon and gets back the results. The use of this plug-in is as follows:

```
merlin localhost 2507 input_file.mrl output_file.mrl
```

Try the last one if you want to run a simulation based on a configuration that is already in the example database.

Analyzing The Results

The result of a simulation is a text file containing a description of the topology and parameters values characterizing the algorithms. Not all the parameters have a meaning for every type of algorithm. These parameters are the following:

AVERAGE_HOPS

For a routing algorithm, this is the average number of hops of each computed route. For the other type of algorithm, the value is either 0 or the value taken by a preceding routing algorithm.

USED_WAVELENGTH

This the total number of wavelength used in the wavelength allocation. An wavelength allocation algorithm tend to minimize this value. It has no meaning for a routing algorithm and take the value 0.

THROUGHPUT

For a routing algorithm, this is the total number of established routes. For the other types of algorithm, it has no meaning and the value is either 0 or the value taken by a preceding routing algorithm.

UTILIZATION

For a routing algorithm, it has no meaning and takes the value 0. For a wavelength allocation algorithm this is given by the following formula:

total number of lightpath / number of links in the topology

BLOCKING_PROBABILITY

For a routing algorithm, this is the number of connections for which the algorithm did not find a route. For a wavelength allocation algorithm, this is given by the following formula :

total number of connections requested - number of established routes.

WAVELENGTH_REUSABILITY

for a wavelength allocation algorithm this is given by the following formula :

number of established routes / number of colors used.

COMPUTATION_COMPLEXITY

This is the time (in seconds) used by the algorithm in order to compute the results.

COST

The cost is computed by a cost function located in each route object. Currently, this function return the number of links used by the route.

WAVELENGTH_AVAILABILITY

This parameter is currently not used.

CHAPTER 3

Configuration File

This chapter describes the syntax of a MERLiN configuration file. It gives to the user the possibility to create his own topology and connection requests. In order for MERLiN to work the user must feed the core with a script file containing some instructions. The core parses the script in order to retrieve the information, and if the syntax is correct, performs the computation of the algorithms. Programming techniques that allow the user the user to modify the syntax of the script file are beyond the scope of this section and will be described in more details in *Part2, Chapter 3: MERLiN's Grammar*. It is important to note that the syntax of a MERLiN script file is close to C programming language syntax.

The File Structure describes the required main components of a script file.

Describing The Components introduces the component used in a topology and they descriptive parameters in the script file.

Describing The Actions describes how to make a request for an algorithm in a script file.

Example provides a working example of a MERLiN file that is found in the archive.

File Structure

In order for the MERLiN core to build its internal structure, a MERLiN script file MUST include two major components:

- A TOPOLOGY,
- And an ACTION block.

Onces these two blocks are defined, each must contains the description of the differents elements composing the user's request.

Topology

The topology part contains the description of the network virtual configuration. It describes the nodes, the links and the routes. The topology block is declared as follows:

```
TOPOLOGY
{
}
```

Action

The action part contains the description of the algorithms to launch using the previously described network topology. The description of an algorithm contains the algorithm's name, the machine location, and a set of parameters depending on the type of algorithm. The action part is declared as follow:

```
ACTION
{
}
```

Describing The Components

The *TOPOLOGY* part contains the definition of all the components of the virtual network. Each component type is grouped into a block. There are six block categories:

- NODE,

- LINK,
- CONNECTION,
- ROUTE,
- WAVE.

Node

The node block groups all the nodes of the user's virtual topology. The syntax is:

```
[NODE_BLOC]
{
   ;
}
```

Inside this block, the characteristics of the nodes must be defined. Two types of nodes are present in MERLiN:

- CROSS-CONNECT CONVERT,
- CROSS-CONNECT NON-CONVERT.

Each one of these types of node contains a set of attributes (or Qos parameters) and a switch table indicating how the wavelengths are switched at the nodes. The attributes of a node are:

BER Bit error rate generated by the node. It is a float value.

DELAY Delay generated by the node.It is a float value.

JITTER Jitter generated by the node.It is a float value.

BW Bandwidth available on this node. It is a float value.

OP_TYPE This flag indicates if the node is used as protection or as normal. It can take two values: NORMAL or PROTECTION.

MODE This flag indicates if the node is used or not. It can take two values: USED or NOT_USED

STATUS This flag indicates if the node is failed or not. It can take two values: FAILED or WORKING.

SWITCH_TABLE Every line in this table represents a connection. When a connection is established between two nodes, a line is created in the switch table of each node crossed by this connection.

Here is an example of two different nodes declaration inside a node block:

```
[NODE_BLOC]
{
   OXC_CONVERT OXC1
   {
            BER 0.2 ;
            DELAY 0.1 ;
            JITTER 0.05 ;
            BW 155 ;
            OP_TYPE PROTECTION ;
            MODE NOT_USED ;
            STATUS WORKING ;
            SWITCH_TABLE
            {
            ;
            }
   }
   OXC_NON_CONVERT OXNC1
   {
            BER 0.1 ;
            DELAY 0.2 ;
            JITTER 0.07 ;
            BW 135 ;
            OP_TYPE NORMAL ;
            MODE NOT_USED ;
            STATUS WORKING ;
            SWITCH_TABLE
            {
            ;
            }
   }
}
```

Another way to declare nodes is to use a default statement and short definitions of nodes. A default statement describes the default values to use in the component declaration.

```
DEFAULT OXC_CONVERT
{
    BER       0.2 ;
    DELAY     0.1 ;
    JITTER    0.05 ;
    BW        155 ;
    OP_TYPE   PROTECTION ;
    MODE      NOT_USED ;
    STATUS    WORKING ;
    SWITCH_TABLE
    {
    ;
    }
}
OXC_CONVERT OXC1 ;
OXC_CONVERT OXC2 ;
```

The attributes of nodes OXC1 and OXC2 defined with the short definition uses values sat by the settings.

Link

The link is a trunk of fiber where each fiber contains a number of wavelength. and lambdas connecting two different nodes. Each link MUST be declared inside a link block and the syntax of a such block is as follows:

```
[LINK_BLOC]
{
    ;
}
```

The following attributes are used to describe a link:

DISTANCE Physical length of the link. Its value is a float.

NB_AMPLIFIER Number of amplifiers along the link. Its value is an integer.

NB_REGENERATOR Number of regenerators along the link. Its value is an integer.

FIBER_RATIO Number of fibers composing the link. Its value is an integer.

LAMBDA_RATIO Number of wavelengths per fiber. Its value is an integer.

BER Bit error rate generated by the link. Its value is a float.

DELAY Delay generated by the link. Its value is a float.

JITTER Jitter generated by the link.Its value is a float.

BW Bandwidth available on this link.Its value is a float.

OP_TYPE This flag indicates if the link is used as protection or as normal. It can take two values: NORMAL or PROTECTION.

MODE This flag indicates if the link is used or not. It can take two values: USED or NOT_USED.

STATUS This flag indicates if the link is failed or not. It can take two values: FAILED or WORKING.

LAMBDA_LIST This list contains the wavelengths which have not the same state, mode or operational type as the the link. Typically, the wavelengths used by a connection or failed when the link is working are in this list.

Here is an example of the declaration of a link declaration:

```
[LINK_BLOC]
{
   LINK OXC1_OXC2 OXC1 OXC2
   {
             DISTANCE 2 ;
             NB_AMPLIFIER 10 ;
             NB_REGENERATOR 10 ;
             FIBER_RATIO 10 ;
             LAMBDA_RATIO 4 ;
             BER0.4 ;
             DELAY 0.3 ;
             JITTER 0.05 ;
             BW 135 ;
             OP_TYPE NORMAL ;
             MODE NOT_USED ;
             STATUS WORKING ;
             LAMBDA_LIST
             {
             2 NORMAL NOT_USED FAILED  ;
             3 NORMAL USED WORKING ;
             12 PROTECTION NOT_USED FAILED  ;
             }
   }
}
```

Similar to a node declaration a link can be declared using default settings:

```
DEFAULT LINK
{
   DISTANCE  2 ;
   NB_AMPLIFIER10 ;
   NB_REGENERATOR10 ;
   FIBER_RATIO 10 ;
   LAMBDA_RATIO 4 ;
   BER       0.4 ;
   DELAY     0.3 ;
   JITTER    0.05 ;
   BW        135 ;
   OP_TYPE   NORMAL ;
   MODE      NOT_USED ;
   STATUS    WORKING ;
   LAMBDA_LIST
   {
            10 NORMAL NOT_USED FAILED  ;
            33 NORMAL USED WORKING ;
            42 PROTECTION NOT_USED FAILED  ;
   }
}
LINK OXC1_OXC2 OXC1 OXC2 ;
```

In this example, link OXC1_OXC2 connects nodes OXC1 and OXC2 and takes the attributes of the default declaration.

Connection

A connection is defined in a connection block. It represents a lightpath established between two nodes. A connection is associated with a *ROUTE* in MERLiN as it is described in the *Part2*, *Chapter 3: Component Hierarchy*. A connection block is defined as follows:

```
[CONNECTION_BLOC]
{
   CONNECTION C1
   {
            START OXC1 ;
            DEST OXC2 ;
            DISTANCE 3 ;
            BER 0.4 ;
```

```
                DELAY 0.3 ;
                JITTER 0.04 ;
                BW 120 ;
     }
}
```

The attributes of a connection are the starting and ending node and some restrictive parameters used for the QoS:

START The source node name.

DEST The destination node name.

DISTANCE The distance requested for this connection between the source and destination nodes.

BER The bit error rate requested for this connection.

DELAY The delay requested for this connection.

JITTER The jitter requested for this connection.

BW The bandwidth requested for this connection.

No default tag has been defined for this type of block.

Route

A route is declared in the route request block. Usually, a route is generated by a routing algorithm for a given connection. A route consists of all the link and nodes that a lightpath traverses between a source and destination pair. The route structure in MERLiN contains a list of nodes, links and their attributes. Note that these attributes are output parameters since they are returned by the routing algorithm. The attributes of a route are as follows:

The meaning of a such structure is te determine if there is a possible physical way to cross the network between two nodes. It contains a list of nodes and some attributes calculated from the nodes and the links crossed by the route. The attributes of a route are:

CONNECTION The name of the associated connection.

DISTANCE The physical distance of the route computed as the sum of the distances of the links on the route.

BER — Bit error rate; its value is calculated from the ber of the links and the nodes crossed by the route. It is computed as follows:

DELAY — end-to-end delay between the source and destination pair. This delay is the sum of the links propagation time and any delay incurred at the nodes.

JITTER — The jitter is derived from the jitter of the nodes and links on the route. It is computed as follows:

BW — The bandwidth calculated from the bandwidth of the nodes and the links on the route. It is computed as follows:

The declaration in the script file of a route is as follows:

```
[ROUTE_BLOC]
{
   ROUTE ROUTE_1
   {
            CONNECTION OXC1_OXC2 ;
            DISTANCE 3 ;
            BER 0.4 ;
            DELAY 0.3 ;
            JITTER 0.04 ;
            BW 120 ;
            NODE_NAME_LIST OXC1 OXC3 OXC4 OXC2 ;
   }
}
```

Wave

A wave is declared in the wave block. For each lightpath established between a source and destination node, there needs to be an associated wavelength that is defined on each link traversed.A wave object indicates which wavelength is used on each link. The attributes of a Wave are:

ROUTE — The route associated to this ligthpath.

WL_ASSIGNED_LIST — The list of the wavelength assigned to this ligthpath.

In the list of wavelength assigned, the structure is reduced to two fields:

LINK — The name of the link where the lambdas are used

LAMBDA The list of lambdas used by the ligthpath on the previously declared link.

A wave object is declared as follow:

```
[WAVE_BLOC]
{
   WAVE
   {
            ROUTE ROUTE_1 ;
            WL_ASSIGNED_LIST
            {
            {
            LINK OXC1_OXC3 ;
            LAMBDA2 ;
            }
            {
            LINK OXC3_OXC4 ;
            LAMBDA2 ;
            }
            {
            LINK OXC4_OXC2 ;
            LAMBDA 2 ;
            }
            }
   }
}
```

The Actions

In order for MERLiN to apply an algorithm to the previously described topology, the user needs to indicate in the action block a list of algorithms to launch. Each action tag is different depending on the action performed, but they all obey the same syntax rules for the attributes:

ACTION It is a tag that contains a description of the action and a set of parameters.

ALGO_LOCATION This attribute has two values separated by a space. The first is the IP address (or name) of the machine where the daemon is running, and the second is the location of the algorithm, relative to the environment variable

MERLINPATH. The concatenation of the environment variable MERLINPATH and algo_location should give an absolute path. This path is used by the daemon to locate the algorithm, so it is absolutely necessary to set the MERLINPATH variable correctly(see *Getting Started*).

LIST OF PARAMETERS Each algorithm uses a set of parameters, that is different depending on the type of the algorithm.

Here is an example for the CREATE_ROUTE action (used to launch the routing algorithms):

```
ACTION
{
   CREATE_ROUTE
   {
            ALGO_LOCATIONlocalhost/algorithm/routing/
shortest_path ;
            CONNECTION C1
            {
            START OXC1 ;
            DEST OXC2 ;
            DISTANCE 3 ;
            BER 0.04 ;
            DELAY 0.5 ;
            JITTER 0.1 ;
            BW 3.5 ;
            }
   }
}
```

Here is the list of all possible actions:

CREATE_ROUTE

ASSIGN_WAVELENGTH

RECONFIGURATION

ADD_CONNECTION

REM_CONNECTION

LINK_FAILURE

NODE_FAILURE

Some algorithms have been developed for the first three actions. Here are examples showing how to declare them:

CREATE_ROUTE:

```
    CREATE_ROUTE {
      ALGO_LOCATION localhost /algorithm/routing/shortest_path
;
         CONNECTION C2 {
          START    OXC1 ;
          DEST     OXC3 ;
          DISTANCE 5 ;
          BER      6 ;
          DELAY    7 ;
          JITTER   8 ;
          BW     9.3 ;
      }
    }
```

ASSIGN_WL

```
    ASSIGN_WL {
      ALGO_LOCATION localhost /algorithm/wavelength/tabu ;
        CONNECTION C2 ;
    }
```

RECONFIGURATION

```
    RECONFIGURATION {
          ALGO_LOCATION  localhost  /algorithm/reconfiguration/
first_fit_reconf
             CONNECTION C2 {
               START    OXC1 ;
               DEST     OXC3 ;
               DISTANCE 5 ;
               BER      6 ;
               DELAY    7 ;
               JITTER   8 ;
               BW     9.3 ;
```

```
        }
        CONNNECTION_MAX 40 ;
        MEAN_CONNECTION_SET 5 ;
      }
```

Example File

```
# A mesh network
# (10 nodes, 13 bidirectional links)
TOPOLOGY {
  # All the nodes are considered the same
  [NODE_BLOC]
 {
    DEFAULT OXC_CONVERT
    {
      BER            0.2 ;
      DELAY          0.1 ;
      JITTER         0.05 ;
      BW             155 ;
      OP_TYPE        NORMAL ;
      MODE           NOT_USED ;
      STATUS         WORKING ;
      SWITCH_TABLE {
      ;
      }
    }
    OXC_CONVERT OXC1 ;
    OXC_CONVERT OXC2 ;
    OXC_CONVERT OXC3 ;
    OXC_CONVERT OXC4 ;
    OXC_CONVERT OXC5 ;
    OXC_CONVERT OXC6 ;
}

#LINK BLOC

  [LINK_BLOC]
  {
    DEFAULT LINK {
      DISTANCE       20 ;
      NB_AMPLIFIER   2 ;
      NB_REGENERATOR 7 ;
      FIBER_RATIO    2 ;
      LAMBDA_RATIO   3 ;
      BER            0.4 ;
      DELAY          0.2 ;
      JITTER         .05 ;
      BW             135 ;
      OP_TYPE        NORMAL   ;
      MODE           NOT_USED ;
      STATUS         WORKING  ;
      LAMBDA_LIST {
       ;
      }
    }
    LINK OXC1_OXC2 OXC1 OXC2 {
      DISTANCE       2 ;
      NB_AMPLIFIER   2 ;
      NB_REGENERATOR 7 ;
      FIBER_RATIO    2 ;
      LAMBDA_RATIO   3 ;
      BER            0.4 ;
      DELAY          0.2 ;
      JITTER         .05 ;
      BW             135 ;
      OP_TYPE        NORMAL   ;
      MODE           NOT_USED ;
      STATUS         WORKING  ;
      LAMBDA_LIST {
       ;
      }
    }
    LINK OXC1_OXC4 OXC1 OXC4 {
      DISTANCE       5 ;
      NB_AMPLIFIER   2 ;
      NB_REGENERATOR 7 ;
      FIBER_RATIO    2 ;
      LAMBDA_RATIO   3 ;
      BER            0.4 ;
      DELAY          0.2 ;
      JITTER         .05 ;
      BW             135 ;
      OP_TYPE        NORMAL   ;
      MODE           NOT_USED ;
      STATUS         WORKING  ;
      LAMBDA_LIST {
       ;
      }
    }
    LINK OXC1_OXC3 OXC1 OXC3 {
      DISTANCE       16 ;
      NB_AMPLIFIER   2 ;
      NB_REGENERATOR 7 ;
      FIBER_RATIO    2 ;
      LAMBDA_RATIO   3 ;
      BER            0.4 ;
      DELAY          0.2 ;
      JITTER         .05 ;
      BW             135 ;
      OP_TYPE        NORMAL   ;
      MODE           NOT_USED ;
      STATUS         WORKING  ;
      LAMBDA_LIST {
       ;
      }
    }
    LINK OXC1_OXC6 OXC1 OXC6 {
      DISTANCE       9 ;
      NB_AMPLIFIER   2 ;
      NB_REGENERATOR 7 ;
      FIBER_RATIO    2 ;
      LAMBDA_RATIO   3 ;
      BER            0.4 ;
      DELAY          0.2 ;
      JITTER         .05 ;
      BW             135 ;
      OP_TYPE        NORMAL   ;
      MODE           NOT_USED ;
      STATUS         WORKING  ;
      LAMBDA_LIST {
       ;
      }
    }
    LINK OXC2_OXC3 OXC2 OXC3 {
      DISTANCE       3 ;
      NB_AMPLIFIER   2 ;
      NB_REGENERATOR 7 ;
      FIBER_RATIO    2 ;
      LAMBDA_RATIO   3 ;
      BER            0.4 ;
      DELAY          0.2 ;
      JITTER         .05 ;
      BW             135 ;
      OP_TYPE        NORMAL   ;
      MODE           NOT_USED ;
      STATUS         WORKING  ;
      LAMBDA_LIST {
       ;
      }
    }
    LINK OXC2_OXC4 OXC2 OXC4 {
      DISTANCE       2 ;
      NB_AMPLIFIER   2 ;
      NB_REGENERATOR 7 ;
      FIBER_RATIO    2 ;
      LAMBDA_RATIO   3 ;
      BER            0.4 ;
      DELAY          0.2 ;
      JITTER         .05 ;
      BW             135 ;
      OP_TYPE        NORMAL   ;
      MODE           NOT_USED ;
      STATUS         WORKING  ;
      LAMBDA_LIST {
       ;
      }
    }
    LINK OXC3_OXC2 OXC3 OXC2 {
      DISTANCE       4 ;
      NB_AMPLIFIER   2 ;
      NB_REGENERATOR 7 ;
      FIBER_RATIO    2 ;
      LAMBDA_RATIO   3 ;
      BER            0.4 ;
      DELAY          0.2 ;
      JITTER         .05 ;
      BW             135 ;
      OP_TYPE        NORMAL   ;
      MODE           NOT_USED ;
      STATUS         WORKING  ;
      LAMBDA_LIST {
       ;
      }
    }
    LINK OXC4_OXC2 OXC4 OXC2 {
      DISTANCE       2 ;
      NB_AMPLIFIER   2 ;
      NB_REGENERATOR 7 ;
      FIBER_RATIO    2 ;
      LAMBDA_RATIO   3 ;
      BER            0.4 ;
      DELAY          0.2 ;
      JITTER         .05 ;
      BW             135 ;
      OP_TYPE        NORMAL   ;
      MODE           NOT_USED ;
      STATUS         WORKING  ;
      LAMBDA_LIST {
       ;
      }
    }
    LINK OXC4_OXC3 OXC4 OXC3 {
      DISTANCE       10 ;
      NB_AMPLIFIER   2 ;
      NB_REGENERATOR 7 ;
      FIBER_RATIO    2 ;
      LAMBDA_RATIO   3 ;
      BER            0.4 ;
      DELAY          0.2 ;
      JITTER         .05 ;
      BW             135 ;
      OP_TYPE        NORMAL   ;
      MODE           NOT_USED ;
      STATUS         WORKING  ;
      LAMBDA_LIST {
       ;
      }
    }
    LINK OXC4_OXC5 OXC4 OXC5 {
      DISTANCE       8 ;
      NB_AMPLIFIER   2 ;
      NB_REGENERATOR 7 ;
      FIBER_RATIO    2 ;
      LAMBDA_RATIO   3 ;
      BER            0.4 ;
      DELAY          0.2 ;
      JITTER         .05 ;
      BW             135 ;
      OP_TYPE        NORMAL   ;
      MODE           NOT_USED ;
      STATUS         WORKING  ;
      LAMBDA_LIST {
       ;
      }
    }
    LINK OXC5_OXC3 OXC5 OXC3 {
      DISTANCE       9 ;
      NB_AMPLIFIER   2 ;
      NB_REGENERATOR 7 ;
      FIBER_RATIO    2 ;
      LAMBDA_RATIO   3 ;
      BER            0.4 ;
      DELAY          0.2 ;
      JITTER         .05 ;
      BW             135 ;
      OP_TYPE        NORMAL   ;
      MODE           NOT_USED ;
      STATUS         WORKING  ;
      LAMBDA_LIST {
       ;
      }
    }
    LINK OXC6_OXC2 OXC6 OXC2 {
      DISTANCE       10 ;
      NB_AMPLIFIER   2 ;
      NB_REGENERATOR 7 ;
      FIBER_RATIO    2 ;
      LAMBDA_RATIO   3 ;
      BER            0.4 ;
      DELAY          0.2 ;
      JITTER         .05 ;
      BW             135 ;
      OP_TYPE        NORMAL   ;
      MODE           NOT_USED ;
      STATUS         WORKING  ;
      LAMBDA_LIST {
       ;
      }
    }
  }
  [CONNECTION_BLOC]
  {
      ;
  }
```

```
  [ROUTE_BLOC]
  {
   ;
  }

  [WAVE_BLOC]
  {
   ;
  }
}

ACTION {

    CREATE_ROUTE {
      ALGO_LOCATION localhost /algorithm/routing/shortest_path
;
         CONNECTION C2 {
          START    OXC1 ;
          DEST     OXC3 ;
          DISTANCE 5 ;
          BER      6 ;
          DELAY    7 ;
          JITTER   8 ;
          BW     9.3 ;
      }
    }

    ASSIGN_WL {
      ALGO_LOCATION localhost /algorithm/wavelength/tabu ;
        CONNECTION C2 ;
    }
  }
```

CHAPTER 4

Graphical User Interface

Getting started

The Graphical User Interface (GUI) of the MERLiN tool has been built in order to help the user creating small configuration files and quickly analyze the performance results of the algorithms. This section provides the user with a complete reference of the possibilities offered by the GUI.

Getting started gives basic instructions concerning the installation process and the requirements in order to run the interface, and an overview of the interface.

Creating a configuration provides some information in order to quickly create a topology, and run simulation of the configuration and algorithm selected in MERLiN,

Open an existing configuration describes an opening file and its associated dialog box,

Topology information panel explains manipulating of the display information, and describes all the fields of the information panels,

Customizing the document display gives basic tips in order to use the display or make it more readable,

Saving a configuration file analyzes scripts by saving or exporting results.

Installation requirements

The GUI has been developed as a plug-in. It is an external application using the resources offered by the MERLiN tool. Thus the MERLiN server should be installed on your local computer, or on a distant machine in order to perform and retrieve the information from a simulation. Please make sure that MERLiN has been correctly installed by following the installation procedures explained in the corresponding Chapter of this user's manual.

In order to be compatible with most platforms, the application is developed using the Java technology and compilers. You have to make sure that every component of the Java compiler and development kit are present on your local computer. The GUI has been developed and tested under the Java Development Kit (Jdk) 1.1.7, and the compliance has not been tested with later versions. The Java Swing 1.1.1 package is also needed in order to run the interface, and the compatibility with later versions is not ensured as well. Please refer to the specific installation documentation of these products in order to get the installation and compatibility information. For convenience, you should be able to find some information and downloads from the following URL:

Java Development Kit - http://java.sun.com/products/jdk
Java Swing - http://java.sun.com/products/jfc

First, make sure that all the components needed to build the interface are installed and working. If you are running the Unix version, the GUI should be located where all the plug-ins have been installed (when performing the MERLiN installation process). To run the GUI under Windows, download the directory gui from the unix version to your local computer. Once this operation is performed, change your current directory to the GUI directory and check if the file *"Gui.class"* and *"Merlin-GUI.jar"* are present. If any of these file are missing then the installation process will fail.

If you have downloaded the entire MERLiN archive on your local PC, and you have the Jdk and Swing installed on your Windows system, you can compile the GUI directly at the MS-DOS prompt. Go to the GUI source directory located in *"merlin++/src/plug-in/gui"* and enter:

Batch\Comp

This will compile the GUI and if no error occurs, it will make all the objects required to run the application on your PC.

How to launch the Graphical User Interface?

The MERLiN daemon must be running when launching the GUI in order to retrieve the list of algorithms available in the database. If the MERLiN server is not available, the GUI will still be able to work but in a limited version. You will not be able to send any requests, and to simulate any configuration file. You must be in the directory where the GUI has been installed, and if the daemon is installed and running properly, you can enter in your shell or MS-DOS prompt the following command:

java Gui ***servername***

where *"servername"* can be either localhost if you are running the server on the same machine than the interface, or a distant machine's name, or even the IP address of the MERLiN server machine. Examples, *java Gui* ***localhost***, *java Gui* ***myhost.nist.gov***, *java Gui* ***149.140.60.5***. If the GUI can not reach the MERLiN server for any reason, you will get a warning message or an error message in your shell or MS-DOS prompt. Please see the troubleshooting section in order to solve the common problems encountered by the users. After Java has initialized all the components and objects of the interface, the interface presented in Figure 6 appear on your screen.

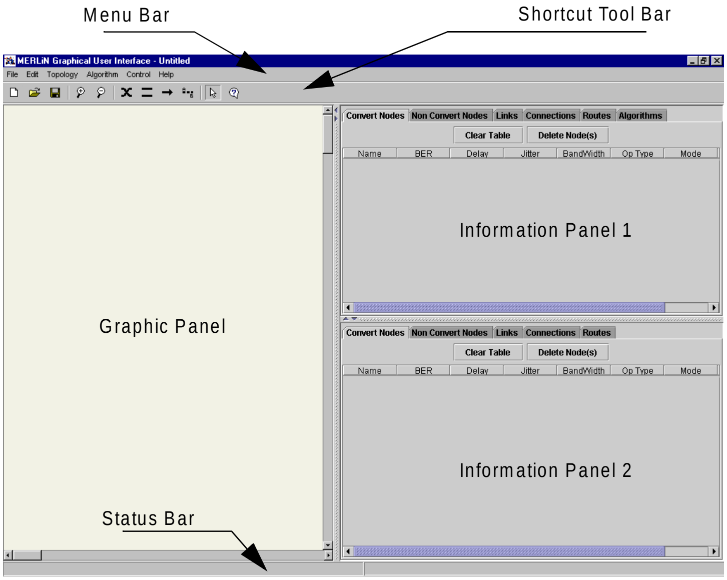


FIGURE 6. **Graphical User Interface overview**

Shortcut Tool Bar

The shortcut tool bar is a set of icons that helps to do common tasks quicker. The different tasks have been divided into four categories: the file management, the display settings, the topology creation and the component information as shown in Figure 7. Most of the icons are exclusive in their own category, and the use of one removes the selection of the others.

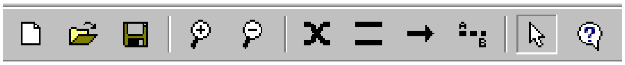


FIGURE 7. **Shortcut Tool Bar**

File management

Erase the current topology, and start over from a blank screen. This shortcut is equivalent to the menu File>New.

Open and load an existing document. This shortcut is equivalent to the menu File>Open.

Save the current configuration. This shortcut is the equivalent of the menu File>Save.

Display Settings

Zoom In the current displayed topology. This shortcut correspond to the Display>Zoom In menu.

Zoom Out the current displayed topology. This shortcut is identical to the Display>Zoom Out menu.

Topology Creation

Create a new Cross Connect with wavelength conversion. Same as the menu Edit>Create>Cross Connect Convert.

Add a new Cross Connect without wavelength conversion. Equivalent to Edit>Create>Cross Connect Non Convert.

Create a new Link between two nodes. Equivalent to the menu Edit>Create>Link.

Create a new Connection request between two nodes. Equivalent to the menu Edit>Create>Connection.

Component Information

Node selection. Select and allow the user to move nodes inside the topology displayed.

Node information. Select and add in the information panel all the nodes inside the boundary box drawn. It will add automatically in the other panel the dependencies of the node (like the associated links, connections,...)

Thumbnails Overview

The thumbnails are located in the information panels. These are designed in order to observe and modify the information of the components, algorithms and results currently displayed in the interface.

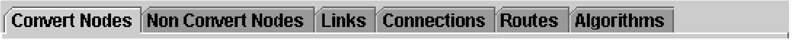
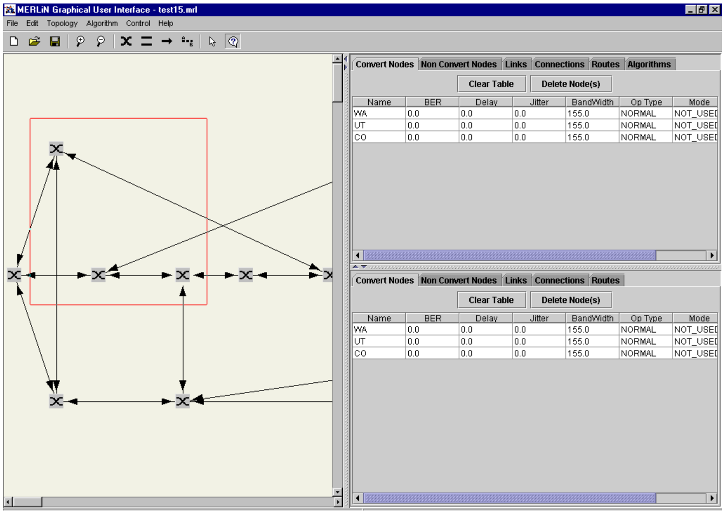


FIGURE 8. **Thumbnails overview**

As presented in the Figure 8, each thumbnail is clickable and displays some information on the configuration. A full description of each field and its meaning is provided in the *Topology Information Panel* section of this Chapter. Two thumbnails panels have been implemented in the interface in order for the user to compare related information of the current topology.

Getting/Setting information

Use the *“Information”* shortcut icon in order to select the components to display in the thumbnail panels. Once the information button is selected, click on one node in order to add it to the list of selected nodes. If you want to select several nodes, left click on a blank portion of the graphical zone, and drag the mouse. A red box should appear following the mouse movements as shown in Figure 9.



Name	BER	Delay	Jitter	BandWidth	Op Type	Mode
WA	0.0	0.0	0.0	155.0	NORMAL	NOT_USED
UT	0.0	0.0	0.0	155.0	NORMAL	NOT_USED
CO	0.0	0.0	0.0	155.0	NORMAL	NOT_USED

Name	BER	Delay	Jitter	BandWidth	Op Type	Mode
WA	0.0	0.0	0.0	155.0	NORMAL	NOT_USED
UT	0.0	0.0	0.0	155.0	NORMAL	NOT_USED
CO	0.0	0.0	0.0	155.0	NORMAL	NOT_USED

FIGURE 9. **Information selection on components**

The selected components like the nodes and links appear in their respective thumbnail panels. To change the information of a field, display the thumbnail panels and go to the line associated with the component to edit. Then click with the left mouse button on the field, and enter a new numeric value or name. Some fields have restricted choices that will appear as a list of choices. Select one of them and it will be the new value for the field.

Another way to get some information on the nodes is to use the *“Properties”* option of the floating menu over a node. Select a node of the topology with the mouse pointer and click on the right mouse button. The floating menu presented in Figure 10 will appear.

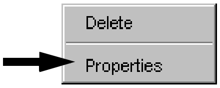
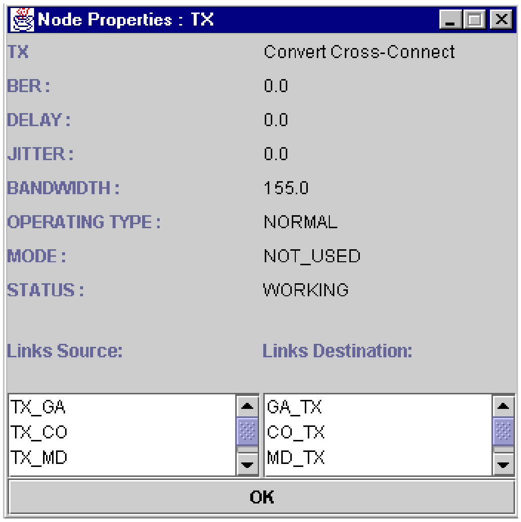


FIGURE 10. **Node Properties**

Once the *"Properties"* option is selected, another window will appear displaying the characteristics of the node. Each field is for information only and can not be changed. Figure 11 represents a typical properties window for a node.



Node Properties : TX

TX	Convert Cross-Connect
BER :	0.0
DELAY :	0.0
JITTER :	0.0
BANDWIDTH :	155.0
OPERATING TYPE :	NORMAL
MODE :	NOT_USED
STATUS :	WORKING

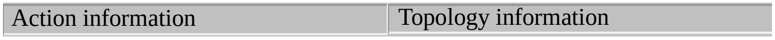
Links Source:	Links Destination:
TX_GA	GA_TX
TX_CO	CO_TX
TX_MD	MD_TX

FIGURE 11. **Properties Window for a Node**

This window can be iconified for later use or dismissed when clicking on the *"Ok"* button (or when closing the window).

Status Bar

Each time you are doing an action in the interface a message is displayed at the bottom of the window, in the status bar. This bar is divided in two parts as shown in the Figure 12.



Action information | Topology information

FIGURE 12. **Status Bar Overview**

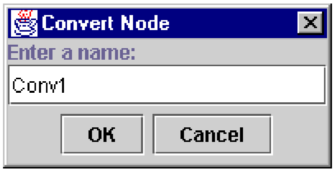
The action information is the result of an action or the type of action the interface is expecting. The topology information is used in order to determine the component that is being pointed to by the mouse. This is useful in order to quickly determine the name of a node when creating a link or connection request.

Creating a configuration

In order to build a topology, it is recommended to first place several nodes on the graphical display, and then connect them and create requests for connections. This section explains step by step the process of building a configuration and editing it.

Creating nodes

Select the type of node you want to create using the menu Edit>Create>Cross Connect Convert or Edit>Create>Cross Connect Non Convert, or the similar icons on the shortcut tool bar. Then click on the graphical zone in order to place a new node. Each time a node is created, the interface will prompt you for a name in a dialog text box as shown in the Figure 13.



Convert Node

Enter a name:

Conv1

FIGURE 13. **Node Name Prompt**

It is important to choose carefully the name of your node, as this one can not be changed later. In order to change the name of a node, you will have to delete it and consequently its associated structures (links, connections, routes,...) and recreate it. Once a node is created, you can move it within the graphical zone by clicking on the *“Selection”* arrow icon of the shortcut tool bar, or modify the different fields by clicking on the *“Information”* icon and change the fields values in the corresponding thumbnail (See *Getting/Setting Information* in this Chapter). If at the prompt for a name you finally do not want to create the node, click on the *Cancel* button or close the window and no changes will be applied to the topology.

Creating Links

Select the menu Edit>Create>Link or the link icon in the shortcut tool bar, then click on the source node for the link. You will note that the status bar will prompt you for a destination node. The last step is to select a destination for the link and click on it. A link going from the source to the destination node should appear on the screen.

Creating Connection Requests

When the full topology is created, and if you wish MERLiN to compute routes or find wavelength assignment, you have to first create some connection requests between nodes.

Select in the menu Edit>Create>Connection or use the dedicated connection icon in the shortcut tool bar. The selection is similar to a link as you have to select the source node first, and then a destination node by clicking on each one. The status bar indicates the node selected. When created, the connection request is displayed in the connection thumbnail panel.

Deleting Nodes

To delete a node, you have to be in the *"Selection"* mode accessible from the shortcut tool bar. Place the mouse on the node you want to delete and right click with the right mouse button. The floating menu presented in the Figure 14 will appear.

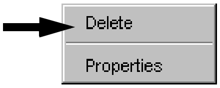


FIGURE 14. **Delete Nodes**

Selecting the Delete option of this menu will remove the node from the graphical zone. When a node is deleted, all the resources and structures associated are destroyed. The links associated are removed from the screen and the thumbnail panels, and connections using the node are removed from the lists. The routes using the node are no longer valid.

Deleting Links

The deletion of a link alone is done in the link thumbnail panel. In order to select a link and display it in the panel, you have to select at least a node connected to the concerned link. Click on the *"Information"* icon of the shortcut tool bar and click on a node connected to the link you want to remove. Then go to one of the information panel and choose to see the link thumbnail panel where you should see a list containing the link that has to be removed. Left click with the mouse on the line you want to remove, and maintain the mouse button a few seconds until the background color of the line changes. You are also allowed to select several lines in the table by maintaining the left mouse button and drag the mouse over the lines you want to delete. Then click on the *"Delete Link(s)"* button on top of the thumbnail panel, and the links will disappear from the list. All the structures associated with the links selected will be removed such as the calculated routes using them. The nodes and connection requests will remain unchanged.

Some key combinations are allowed in order to select several links easily. Selecting a line and maintain the SHIFT key will, on the next left mouse button click select all the links between the link selected and the one currently under the mouse pointer. Using the CTRL key selects non-consecutive links in the list.

Deleting Connection Requests

Deleting a connection request is similar to a link deletion. Display the thumbnail panel associated with the connection in one of the information panel, and select one or several connection to be removed (see the *Deleting Links* paragraph for selection tips). Then click on the *"Delete Connection(s)"* button that will remove the internal structures of the selected connections and the associated routes.

An efficient way to get familiar with the GUI is to start from an already created topology. Some example files are provided with the MERLiN archive, and it is recommended for beginners to modify an existing configuration (See the *Opening an existing configuration* of this Chapter). Try to add more nodes or delete some in order to see the actions associated with the menus and icons.

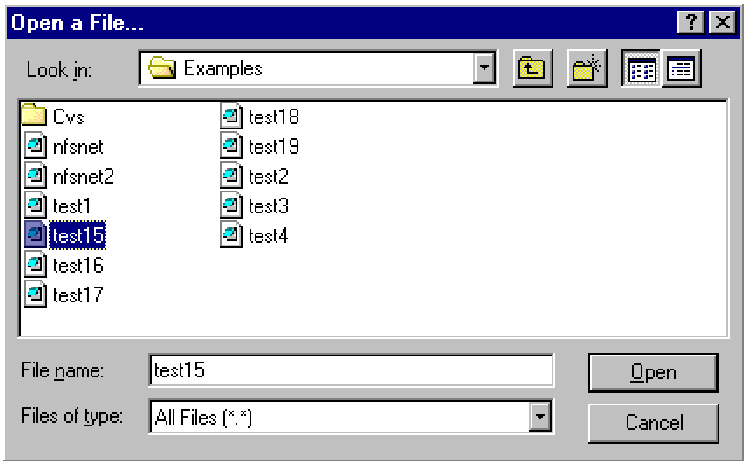
Simulate the configuration in MERLiN

When the topology, connection requests, and algorithm selection are ready, choose the *Control>Start* option in the menu bar in order to send the request to the MER-

LiN server. Once the results are ready, check the thumbnail panels in order to observe the results in the algorithms performance panel and route panel.

Opening an existing configuration

Opening a configuration file can be done using the shortcut tool bar icon or by the menu *File>Open*. Each time a new configuration is loaded, the interface erases the current configuration from the screen and its associated data structures, then the topology is automatically replaced by the new one. Before loading a new configuration, please make sure that your work is saved first. When clicking on the shortcut icon or selecting the option Open in the File menu, a file selection window appears on the screen as presented in Figure 15.



Look in: Examples

File name: test15

Files of type: All Files (*.*)

FIGURE 15. Open File Dialog Box

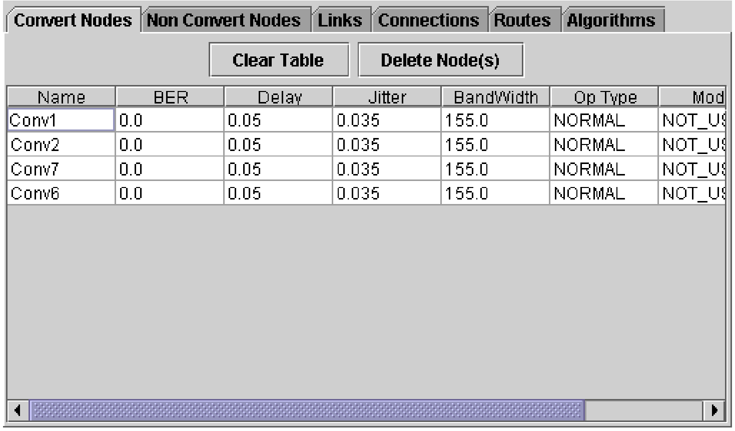
The organization of this dialog box depends on the system you are running. It is recommended to refer to the specific documentation provided with your operating system in order to use the selection box. Browse your hard drive or eventually the network in order to find the MERLiN configuration file you want to load, and select it. When opening a file, the interface will check if it is a valid file then it will display the new topology and its information on the screen. The MERLiN archive provides some example of files that can be loaded in the interface. These files are located in the *“Examples”* directory of the GUI source archive. Feel free to load them and modify them in order to familiarize yourself with the interface.

Topology information panel

The topology information panel has been divided in two similar parts in order for the user to compare different aspects of the configuration at the same time. Each part is divided in several thumbnail panels as seen in Figure 8 on page 55. This section details the different panels and the meaning of the parameter found in it.

Non Convert Node Thumbnail Panel

The non convert node thumbnail panel corresponds to the cross connect with no convertion present and selected in the topology. it is composed of two parts as described in the Figure 16. The top part is to manage the table and the deletion of nodes inside it. The bottom part is the list containing the information of the components. Use the scroll bars to retrieve extra information and browse the list.



Name	BER	Delay	Jitter	BandWidth	Op Type	Mod
Conv1	0.0	0.05	0.035	155.0	NORMAL	NOT_US
Conv2	0.0	0.05	0.035	155.0	NORMAL	NOT_US
Conv7	0.0	0.05	0.035	155.0	NORMAL	NOT_US
Conv6	0.0	0.05	0.035	155.0	NORMAL	NOT_US

FIGURE 16. **Cross-Connect Non Convert Thumbnail Panel**

The different fields composing the list of cross connect are editable. Please refer to the *Getting/Setting information* section in order to edit the information concerning the nodes inside the thumbnail panels.

Each cross-connect has the same set of parameters that is described here:

Name: Conv1

The name of the node. Please note that this name is linked to the link and connection requests name. Once set when creating a new node, this name can not be changed without deleting the node and create a new one.

BER: 0.4

Bit Error Rate of the node.

Delay: 0.5

Delay added by the node.

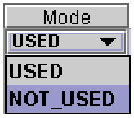
Jitter: 0.0

Jitter introduced by the node.

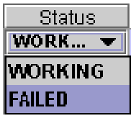
BandWidth: 155.0

Bandwidth of the node.

The type of operation of the node. Normal is for a common utilization. Protection mean that the component is used for protection only. This is mostly used in restoration algorithms when a fault has occurred in the network.



The mode indicates if the node is in use or not. Usually this parameter is automatically set by MERLiN when a route is using the node and some optical switching is required.



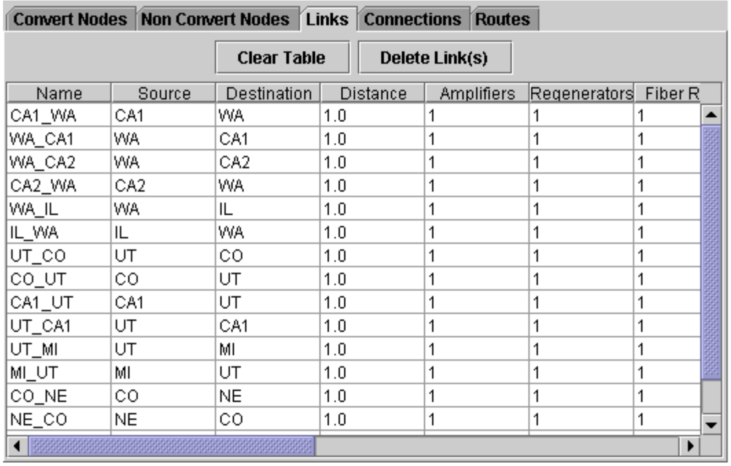
The status of the node is mostly used to study reconfiguration algorithms. Working is the normal state, if the node is set to failed, all the lambdas inside are also failed, meaning no assignment of wavelength can be done, and restoration algorithm need to use the protection components.

Convert Node Thumbnail Panel

The Convert thumbnail panel is similar to the non convert node thumbnail panel. It contains information on the selected cross-connect convert nodes. Please refer to the *Non Convert Node Thumbnail Panel* section above to retrieve the fields. Convert nodes need to have a matrix of convertion that has not been implemented yet.

Link Thumbnail Panel

The link thumbnail panel is composed of two parts as described in Figure 17. The top part is to manage the table and the deletion of links and the bottom part is the list containing the link information. Use the scroll bars to retrieve extra information on the components and browse the list.



Name	Source	Destination	Distance	Amplifiers	Regenerators	Fiber R
CA1_WA	CA1	WA	1.0	1	1	1
WA_CA1	WA	CA1	1.0	1	1	1
WA_CA2	WA	CA2	1.0	1	1	1
CA2_WA	CA2	WA	1.0	1	1	1
WA_IL	WA	IL	1.0	1	1	1
IL_WA	IL	WA	1.0	1	1	1
UT_CO	UT	CO	1.0	1	1	1
CO_UT	CO	UT	1.0	1	1	1
CA1_UT	CA1	UT	1.0	1	1	1
UT_CA1	UT	CA1	1.0	1	1	1
UT_MI	UT	MI	1.0	1	1	1
MI_UT	MI	UT	1.0	1	1	1
CO_NE	CO	NE	1.0	1	1	1
NE_CO	NE	CO	1.0	1	1	1

FIGURE 17. **Link Thumbnail Panel**

Here is a full description of each field present in the panel:

Name
CA1_WA

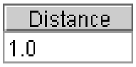
The name of a link is automatically determined by the source and destination node names. When a link is created between two nodes, it is impossible to change the source and/or the destination nodes. In order to change the nodes attached to a link, it is required to delete the link and recreate a new one.

Source
CA1

The source node for the link (where the link starts).

Destination
WA

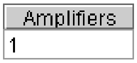
The destination node for the link (where the link ends).



Distance

1.0

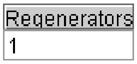
The fiber length.



Amplifiers

1

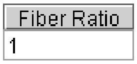
Number of amplifiers used along the link.



Regenerators

1

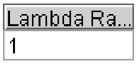
Number of regenerators along the link.



Fiber Ratio

1

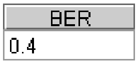
Number of fibers inside the link.



Lambda Ra...

1

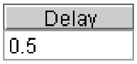
Number of wavelength per fiber.



BER

0.4

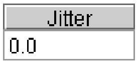
Bit Error Rate of the fiber used in the link. It is assumed that all fibers in the link have the same properties.



Delay

0.5

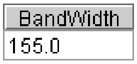
Delay added by the link.



Jitter

0.0

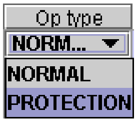
Jitter introduced by the link.



BandWidth

155.0

Bandwidth of the link.



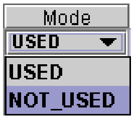
Op type

NORM...

NORMAL

PROTECTION

The Operation Type of this link The link is usually used in normal operation mode unless its operation mode is set to protection. This is used in restoration algorithms to recover from a network failure.



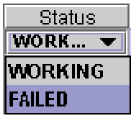
Mode

USED

USED

NOT_USED

The mode indicates if the link is in use or not. Usually this parameter is automatically set by MERLiN. A link mode used means that all the lambdas of the link are used and thus are not available for wavelength allocation.



Status

WORK...

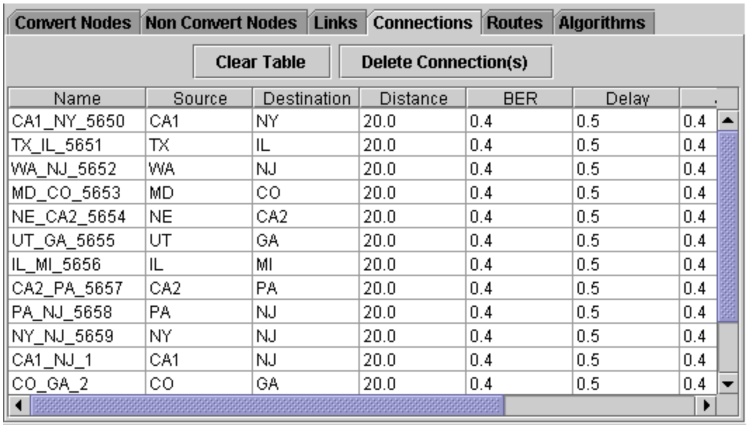
WORKING

FAILED

The status of the link is mostly used to study the restoration. Working is the normal state of a link. If the link is set to failed, all the lambdas inside are also failed, meaning that in some algorithms, no assignment can be done, and the algorithm needs to use protection components.

Connection Thumbnail Panel

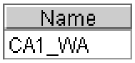
The connection thumbnail contains all the connections requests made on the topology. Similar to the Nodes and Link Thumbnails, the top part is to manage the table and the deletion of connections inside it. The bottom part is the list containing the information on the connections as it is described in the Figure 18. Use the scroll bars to retrieve extra information on the components and browse the list.



Name	Source	Destination	Distance	BER	Delay	.
CA1_NY_5650	CA1	NY	20.0	0.4	0.5	0.4
TX_IL_5651	TX	IL	20.0	0.4	0.5	0.4
WA_NJ_5652	WA	NJ	20.0	0.4	0.5	0.4
MD_CO_5653	MD	CO	20.0	0.4	0.5	0.4
NE_CA2_5654	NE	CA2	20.0	0.4	0.5	0.4
UT_GA_5655	UT	GA	20.0	0.4	0.5	0.4
IL_MI_5656	IL	MI	20.0	0.4	0.5	0.4
CA2_PA_5657	CA2	PA	20.0	0.4	0.5	0.4
PA_NJ_5658	PA	NJ	20.0	0.4	0.5	0.4
NY_NJ_5659	NY	NJ	20.0	0.4	0.5	0.4
CA1_NJ_1	CA1	NJ	20.0	0.4	0.5	0.4
CO_GA_2	CO	GA	20.0	0.4	0.5	0.4

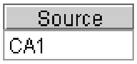
FIGURE 18. **Connection Thumbnail Panel**

Here is a full description for each fields present in the thumbnail panel. The Name, Source and Destination are information in order to build the connection. The other parameters are used by the quality of service algorithms in order to apply some constraints in construction of the connection.



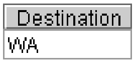
Name
CA1_WA

The name of a connection is automatically determined by the source and destination node names plus a unique id.



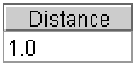
Source
CA1

The source node for the connection request (where the connection starts).



Destination
WA

The destination node for the connection (where the connection ends).



Distance
1.0

The desired distance for this connection (A constraint used by QoS algorithms only).

BER
0.4

The desired Bit Error Rate along the connection (A constraint used by QoS algorithms only).

Delay
0.5

The desired Delay for the connection (A constraint used by QoS algorithms only).

Jitter
0.0

The desired Jitter over the connection (A constraint used by QoS algorithms only).

BandWidth
155.0

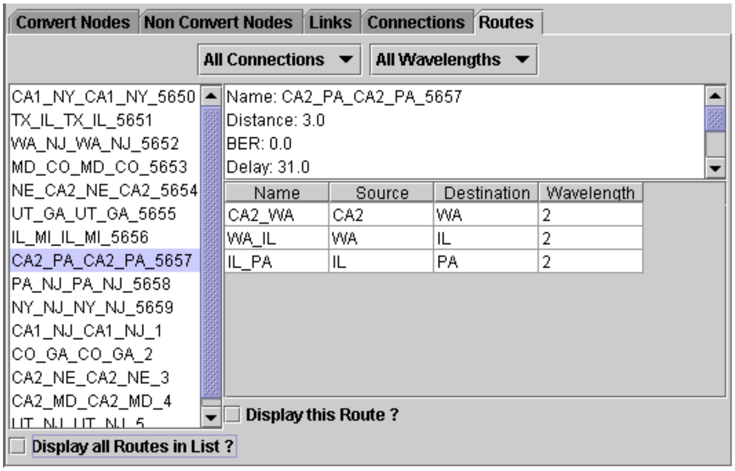
The desired bandwidth for the connection (A constraint used by QoS algorithms only).

Compute ?
false

This flag is set to false if a route and wavelength were assigned for the connection request. The same route and wavelength assignment are used in the case of a reconfiguration algorithm. In case this flag is set to true, a route and wavelength assignment need to be recomputed.

Routes Thumbnail Panel

The route thumbnail panel is used to display the results of a routing algorithm after a simulation is completed.



CA1_NY_CA1_NY_5650
TX_IL_TX_IL_5651
WA_NJ_WA_NJ_5652
MD_CO_MD_CO_5653
NE_CA2_NE_CA2_5654
UT_GA_UT_GA_5655
IL_MI_IL_MI_5656
CA2_PA_CA2_PA_5657
PA_NJ_PA_NJ_5658
NY_NJ_NY_NJ_5659
CA1_NJ_CA1_NJ_1
CO_GA_CO_GA_2
CA2_NE_CA2_NE_3
CA2_MD_CA2_MD_4
UT_NJ_UT_NJ_5

Name: CA2_PA_CA2_PA_5657
Distance: 3.0
BER: 0.0
Delay: 31.0

Name	Source	Destination	Wavelength
CA2_WA	CA2	WA	2
WA_IL	WA	IL	2
IL_PA	IL	PA	2

□ Display this Route ?

□ Display all Routes in List ?

FIGURE 19. **Routes Thumbnail Panel**

Figure 19 on page 67 illustrates the panel and its different parts. There are two list buttons in the top portion of the panel. The connection button is used to choose to display the connections in panel and the wavelengths button is used to select which wavelength to display in the graphical panel. By default, all connections and all wavelengths are displayed.

Each time the user selects a connection in the list located in the left panel, the information specific to the connection is displayed in the right panel. A text box is present in used to retrieve the information on the connection request, and a list of the result parameters is also present. The fields are as follows.

Name: CA1_WA

The name of the connection request.

Source: CA1

The source node for the connection request.

Destination: WA

The destination node for the connection.

Wavelength: 0

The wavelength assigned along the route.

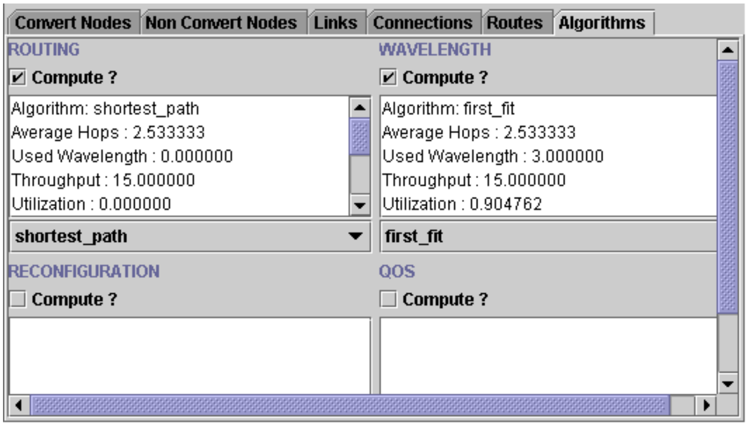
When the check box *“Display this route?”* is checked, the current route is displayed in the graphical panel over the topology, while checking the *“Display all Routes in List?”* check box will display all the routes present in the connection list. Clicking once again on one of the check boxes will remove the drawing of the route(s) from the graphical display.

By using the button located at the top left corner of the thumbnail panel, it is possible to select one or several connection to study. This option will affect the list of connection displayed in the left part of the thumbnail.

Select a specific lambda or display all of them by clicking on the button located on the top right corner of the thumbnail panel. This option will also affect the graphical display panel for the routes displayed on the screen.

Algorithms Thumbnail Panel

The algorithm panel is divided in several categories of algorithms, as shown in Figure 20.



ROUTING

☑ Compute ?

Algorithm: shortest_path
Average Hops : 2.533333
Used Wavelength : 0.000000
Throughput : 15.000000
Utilization : 0.000000

shortest_path

WAVELENGTH

☑ Compute ?

Algorithm: first_fit
Average Hops : 2.533333
Used Wavelength : 3.000000
Throughput : 15.000000
Utilization : 0.904762

first_fit

RECONFIGURATION

□ Compute ?

QOS

□ Compute ?

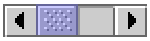
FIGURE 20. **Algorithm Thumbnail Panel**

The category's title is written in blue at the top of each panel and a check box is used in order to enable running the algorithm when the simulation is launched. When the box is checked, the algorithm selected in the list at the bottom part of the algorithm panel will be applied to the topology. It is possible to combine several algorithms for example a Routing and a Wavelength assignment algorithm. When sending a simulation request to the server, the algorithms will be launched one by one in the order of the categories. That is the Routing algorithm will be launched first followed by the Wavelength Assignment algorithm followed by the reconfiguration category. A text box displays the performance results of each algorithm after simulation is complete. If this text information box is empty, the performances have not been calculated yet or the algorithm has not been selected. The list of algorithms at the bottom of the algorithm panel includes all the algorithms available in the category. Choose the one you want to apply before sending the simulation request to MERLiN. In order to save the algorithms performance results, use the Export option (See *Export Performance* of the *Saving A Configuration File* section of this chapter).

Customizing The Document Display

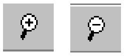
Customizing the display can be an efficient way to study part of a topology, compare the results obtained from simulation, use the different components and adjust the size of the different panels in order to have a clear view of the parameters.

Scroll Bars



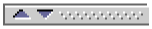
Scroll bars in information panels allow the user to browse the different lists of parameters in the thumbnail panels. In the graphical panel, scroll bars adjust the current display of the topology. This is useful when focusing on a smaller portion of a big topology.

Zoom



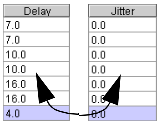
Use the zoom icons in order to study a specific part of a topology, and add details to the current display. The wavelength assignment can only be viewed in some zoom level. When the zoom level is at its minimum, only the routing can be seen at the lowest zoom level.

Split Panels



The split panels allow the user to resize the different parts displayed on the screen. Use it to resize the graphical panel in order to minimize it or maximize it when studying some parameters or drawing a topology. In the information panel it is also possible to remove one of the two thumbnail panels by clicking on the arrows of the split bar.

List Management



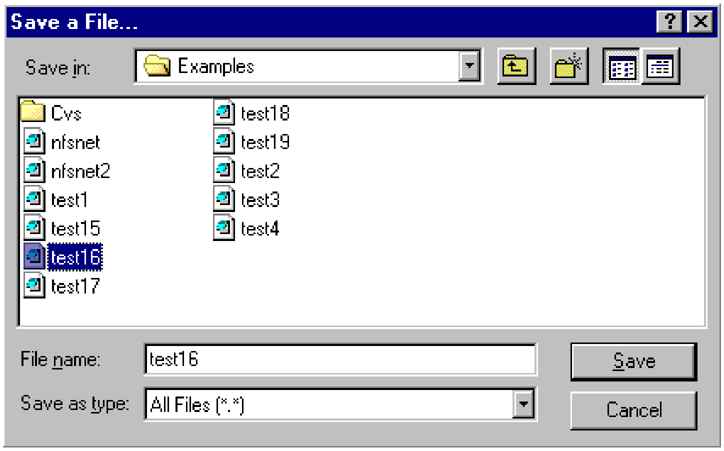
Delay	Jitter
7.0	0.0
7.0	0.0
10.0	0.0
10.0	0.0
16.0	0.0
16.0	0.0
4.0	0.0

Drag and drop the different columns of the lists to place nearby two parameters that you want to study more thoroughly.

Saving A Configuration File

Saving a topology

Saving a configuration file can be done using the shortcut tool bar icon or the menu *File>Save*. When clicking on the shortcut icon or selecting the option Save in the File menu, a file selection window appears on the screen as presented in Figure 21.



Save a File...

Save in: Examples

File name: test16

Save as type: All Files (*.*)

FIGURE 21. **Save File Dialog Box**

The organization of this dialog box can differ depending on the system you are running. It is recommended to refer to the specific documentation provided with your operating system in order to use the selection box. Browse your hard drive or eventually the network in order to locate the directory in which you want to save your configuration file, and pick one file (or give a new name to the file). When selecting an existing file inside a directory, the system will prompt you for an overwrite confirmation. Acknowledge the message if you want to replace an existing configuration or Cancel it if you want to choose another name for your MERLiN file.

Saving a configuration file will save the entire topology description with names and node characteristics, links, some GUI extra information, the connection requests, the algorithms in use, and the routes already computed. However since this file is destined to be the input file for a simulation, it will not include the performance results of the algorithms. In order to retrieve the information on the performance results, it is necessary to export them (see *Export performances*).

Export Performances

The export performances feature provides a way to save the simulation results and use them in external applications like other simulators or analyzing scripts. When exporting algorithm performances, you will be prompt with a window similar to the one for saving a configuration file. Please refer to Figure 21 on page 71 and the *Saving A Configuration File* section of this chapter in order to use the saving dialog box.

Exporting the performances will save the entire topology description with names node characteristics, links, some GUI extra information, the connection requests, the algorithms in use, and the routes already computed. In addition, since this file is destined to be the output file of a simulation, it will include the performance results of the algorithms.

Troubleshooting

This section is destined to give the user some tips in order to solve some of the most common problems that can come up while using the GUI.

The interface hang when loading a configuration

When trying to load a topology containing too much elements in the combo box (lists of the thumbnail panels), typically a huge number of connections (>300) the graphical interface will hang for a lack of memory. This is due to a problem in the Java Swing Package that can not manage more than a certain amount of elements in the lists. To solve the problem, try to reduce the number of connections or split your network in several part to reduce the number of information to manage.

No algorithms are displayed or can be selected

This may be due to various problems.

MERLiN Server is not running

If the MERLiN server can not be reached for any reason (i.e.: the server is not running, or the network is down), it is possible to launch the interface and work under it in order to create a topology and connection requests. However it will not be possible to simulate any algorithms and performance parameters. When the server can not be reached, a warning message is displayed in the current Unix Shell or MS-DOS prompt, and no algorithm is displayed (or can be chosen) in the algorithms menu or panel.

The interface displays a warning concerning the sockets

You must be on the same local network in order for the interface to reach the server. The interface is not built to support Firewall connections, and if you are behind such a gateway and try to access a distant MERLiN server, the interface will not be able to reach it.

The interface does not display the results

After launching a simulation and waiting for some time, if the interface does not display any results. this may means that the connection between the interface and MERLiN is broken. Check with your MERLiN server administrator in order to launch the daemon again or check the network connection.

Part 2: Programmer's Guide

CHAPTER 5

Architecture Overview

This Chapter is dedicated to the internal architecture of MERLiN. The programmer will find here a detailed description of the simulator and how it works.

The Core represents the heart of MERLiN. It is an independant program running on a distant (or local) computer, that dispatches the requests to the different parts of the simulator.

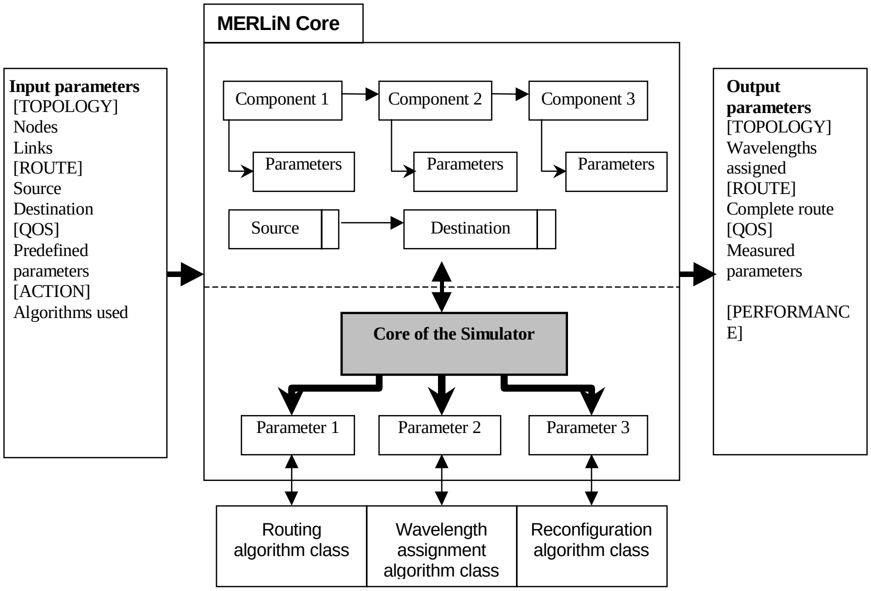
The Plug-in represents an interface module to MERLiN and allows for easy access and extension.

The Algorithm is an essential part in the simulation tool.There are four categories of algorithms to perform routing, wavelength assignment, reconfiguration and QoS support.

The Communication API describes the format of the messages exchanged between the various components during a simulation.

Core

The core of MERLiN is a daemon running in the background and waiting for user's requests. Given an input as a script file that describes the topology, the routes, the Qos parameters and the algorithms to be used, the core builds its own data representation of the network. Then, the core dispatches the topology to the requested algorithms. After the algorithm complets its computation, the core translates its internal structure to a user friendly representation and send the results back to the user.



Plug-in

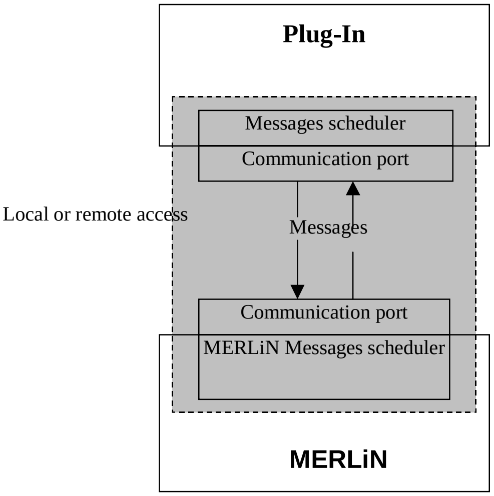
A plug-in is an interface module that allows easy access to MERLiN. It is a standalone application and can be almost anything that communicates with the core in MERLiN's language format. A plug-in acts like a client in MERLiN's client/server architecture. Its functionnality is to send a request to the simulator and collect the result for the user. The interface between the plug-in and the core has been strictly defined and a full description of the language format is given in the *Chapter 3: MERLiN's grammar.*

Algorithm

An algorithm is a special kind of plug-in that communicates with MERLiN's core. The main differences between an algorithm and a plug-in lies in the message exchange : the algorithm uses MERLiN's internal structures while a plug-in uses the script file representation. Another difference is that an algorithm is generally called by the core and not directly by the user.

Communication API

Whenever a communication is needed between a plug-in and the core, or the core and an algorithm, a communication pipe is opened. Then the messages are transfered both ways througth a local or remote port depending on where the different applications are running.



The communication between all the entities composing MERLiN is conducted via a network interface using UNIX sockets. The packets generated by MERLiN have a specific format that is described here. It is not really important to understand this simple format since a template communication API is included in the package to help in building new plug-ins or algorithms.

Size of the packet	MERLiN command	Data

FIGURE 22.

The real advantage of a such interface is to make MERLiN plug-ins and algorithms language independent. As long as the language used supports the sockets communication techniques, and if the MERLiN API is used, the user is free to develop his/her own internal structures and functions, and combine them in any way he/she choose.

CHAPTER 6

Component Hierarchy

Introduction

MERLiN is written in C/C++, using a framework of classes especially designed to simulate WDM networks. The document of reference is the HTML documentation provided with the MERLiN archive. When uncompressing the archive, one can find the source files stored in the *src* directory. The *src* directory contains the following directories :

- **component** contains the classes representing WDM components : links cross-connect, wavelength,etc..
- **math** contains the basic type of data used in MERLiN.
- **merlin** contains the code source of the MERLiN server and some tightly bound to the MERLiN internals.
- **parser** contains the code for the MERLiN file parser and some classes tightly bound to the parser.
- **tool** contains some useful classes like list, array, matrix, vector.
- **network** contains classes used to manage the communication between the different entitties of MERLiN (server, plug-in, algorithms).

The best way to have an overview of the classes available is to look at the HTML documentation.

Hierarchy

Here is a description of hte classe of each category.

Component

The inheritance mechanism has been used only for the design of the *component* classes. The framework provide many level of inheritance in order to ease the development of futur components.

- Component
  this is an abstract class from that every other component is derived.
  Int cid: the id of this component.
  Byte state_mask: status,mode and operation type mask.
  char cname*: name of the component.

- Connection
  this class represents a source-destination pair between two nodes.
  Node src*: source node.
  Node dest*: destination node.
  Int cid: connection id.
  char cname*: connection name.
  Float distance: length of the connection.
  Float ber: bit error rate.
  Float delay: delay for this connection.
  Float jitter: jitter requested by this connection.
  Float bw: bandwidth requested for this connection.
  Int src_id: id of the source node.
  Int dest_id: id of the destination node.

- Lambda
  this class represents a wavelength on a given link.
  Int lid: id of this Lambda.
  Byte state_mask: state mask of this Lambda.

- Link
  this class represents a link between two nodes.
  Float distance: physical distance of this link.
  Int nb_amplifier: number of amplifiers contained along the link.

Int nb_regenerator: number of regenerators contained along the link.
Int fiber_ratio: number of fibers in this link.
Int lambda_ratio: number of wavelengths per fiber.
Float ber: bit error rate.
Float delay: delay introduced by this link.
Float jitter: jitter introduced by this link.
Float bw: bit rate available on this link.
Node node_in*: destination node of the link.
Node node_out*: source node of the link.
lambda_list: list of the wavelengths used in this link.
Int node_in_id: id of the destination node.
Int node_out_id: id of the source node.

- Node
  this is an abstract class that every cross connect is derived from.
  Float m_ber: bit error rate.
  Float m_delay: delay introduced by this node.
  Float m_jitter: jitter introduced by this node.
  Float m_bw: bandwidth of this node.
  List<Link> m_list_link_in: list of the links coming in the node.
  List<Link> m_list_link_out: list of the links coming out the node.

- Oxc
  this is an abstract class that is a generic representation for a cross connect.

- OxcConvert
  this class represents a cross-connect that can do wavelength conversion.

- OxcNonConvert
  this class represents a cross-connect that can not do wavelength conversion.

- Route
  this class represents the ligthpath established for a given connection request.
  Int rid: the route id.
  char rname*: name of the lightpath.
  Float distance: length of the lightpath.
  Float ber: bit error rate of the lightpath.
  Float delay: delay of this ligthpath.
  Float jitter: jitter of this ligthpath.
  Float bw: bit rate of this lightpath.
  Int nb_amplifier: number of amplifiers on this lightpath.

Int nb_regenerator: number of regenerators on this lightpath.
Connection connection*: the connection attached to this lightpath.
List<Node> m_node_list*: list of the nodes along the lightpath.
Int connection_id: id of the connection attached to this lightpath.
List<Int> node_id_list*: list of the id of the nodes along the ligthpath.

- Wave
  this class represents the wavelengths used for a given route.
  *Route*route*: the route attached to this wave.
  List<WlAssigned> wavelength_list*: list of the wavelength used along this ligthpath.
  Int route_id: id of the route attached to this group of wavelengths.

- WlAssigned
  this class represents a set of wavelengths used for one route only.
  Link link:* the link where the wavelengths are assigned.
  List<Lambda> lambda_assigned_list*: the list of the wavelengths assigned on this link.
  Int link_id: the id of the link attached to this set of wavelengths.

The Figure 23 on page 85 show the hierarchy of the Component class.

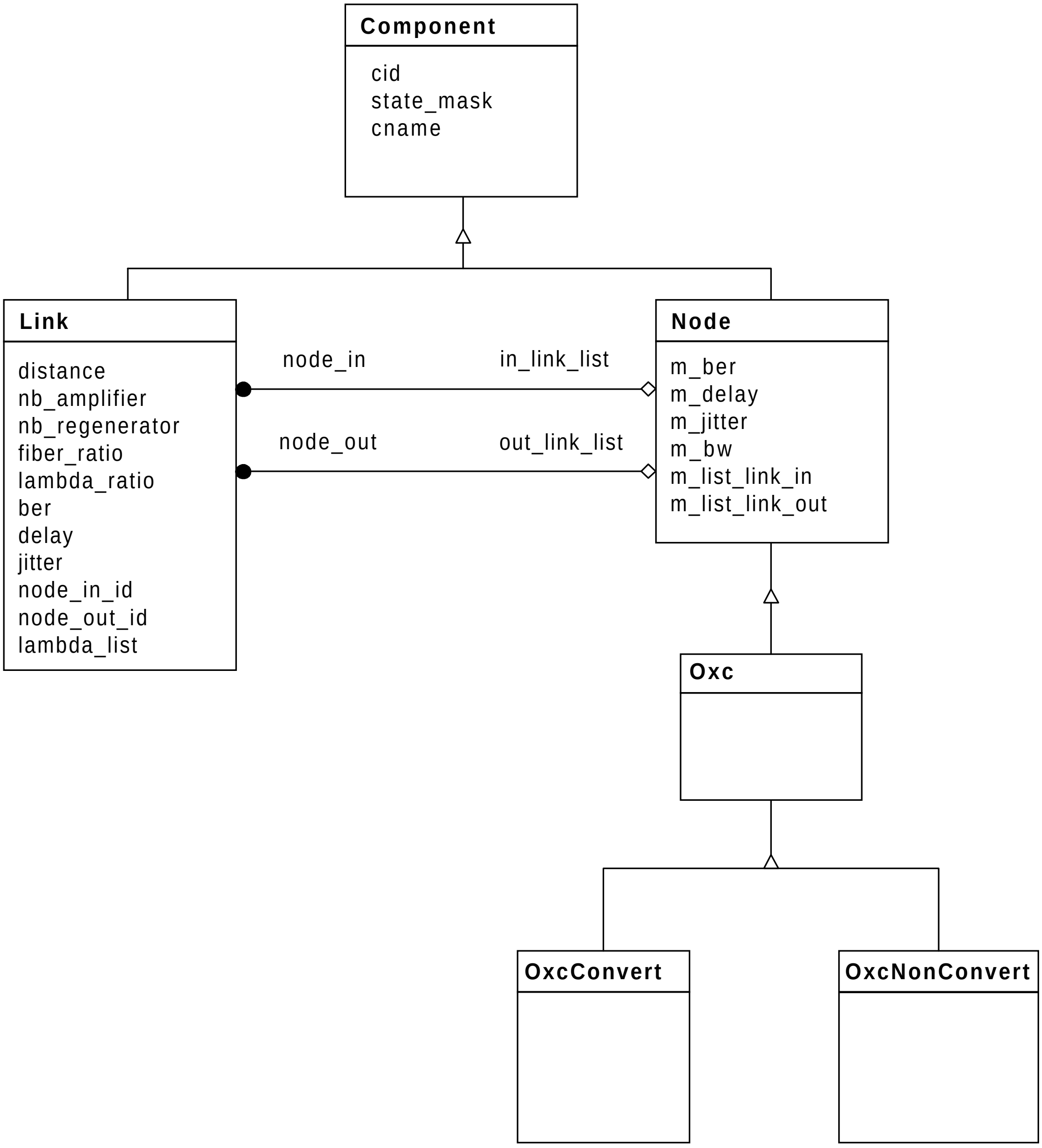


FIGURE 23. hierarchy

Math

Each class of the *math* category is able to transform itself in a common representation in order to avoid architecture-dependent conflict between big-indian and small indian representation systems.

- Boolean
  a class that represents a boolean.
- Byte
  a class that represents a byte. Bytes are heavily used in the communication between the server and the algorithm.
- Double
  a class that represents a double.
- Float
  a class that represents a float.
- Int
  a class that represents an integer.

Merlin

- Parameter
  this class represents a parameter used in the communication between the daemon, the plug-ins and the algorithms.
  Parameter_t type: the type of the *data* attribute of this parameter.
  Int pid: the id of this parameter.
  void data*: the Parameter's data.
- ParameterList
  this class represents a list of instances of Parameter.

Network

- Server
  this class is used in the daemon. There is only one instance of this class.
  Socket server*: the server’s socket.
  Socket client*: the list of Clients connected. Only ONE is connected in this implementation.
  Socket algorithm*: the algorithm currently running.
  boolean is_running: boolean flag that indicates if the server is running or not.
- Client
  this class is used in every client (plug-in or algorithm) that has to connect to the

server.
Socket client*: the socket connected to the server.

- Socket
  this class embedded a subset of the socket API.
  int sid: the system id of this Socket.

Parser

- Algorithm
  this class represents an algorithm. It is used in every algorithm launched by the MERLiN daemon.
  char m_machine*: the name of the algorithm's machine.
  char m_path*: the full path to reach the algorithm.
  Parameter m_parameter*: the input parameter (only available after the parsing of the input file).
  The following attributes are the parameters of the algorithm. They are detailled in “Analyzing The Results” on page 27:
  Float m_average_hops
  Float m_used_wavelengths
  Float m_throughput
  Float m_utilization
  Float m_blocking_probability
  Float m_wavelength_reusability
  Float m_computation_complexity
  Float m_cost
  Float m_wavelength_availability
  Int m_type: the type of the algorithm.
  List<Connection> ml_connection*: list of the connection requests.

- Topology
  this class represents a topology. It contains a list of the nodes, a list of the links, a list of the connections requests and on the ligthpath currently established. Most of the attributes of a Topology object are set after the parsing of the input file or created while calling the copy constructor.
  List<OxcConvert> m_oxc_convert_list*: list of the cross-connect that can do wavelength conversion.
  List<OxcNonConvert> m_oxc_non_convert_list*: list of the cross-connect that can not do wavelength conversion.
  List<Link> m_link_list*: list of the links of the topology.
  List<Connection> m_connection_list*: list of the connection requests.

List<Route> m_route_list*: list of the established lightpaths.
List<Wave> m_wave_list*: list of the wavelengths used by the lightpaths.
List<Algorithm> m_algo_list*: list of the algorithms to start.

Tool

- Array
  this class represents an array. It is mainly used to send the byte representation of the objects through the socket interface.
  Int m_size: size of the array.
  classT m_items*: pointer on the first element of the array.
  Int m_current: index of the current element of the array.
- List
  this class represents a list of pointers on objects.
  Element<classT> head*: pointer on the head of the list.
  Element<classT> current*: pointer on the current element of the list.
  Int elements: number of elements in the list
- Matrix
  this class represents a matrix. It inherits from the Vector class.
- Tool
  this class contains useful functions to be used in the algorithms.
- TopologyMatrix.
  this class is another representation of the topology. It can be more useful in some algorithm.
- Vector
  this class represents a vector. It is the parent class of the class Matrix.

CHAPTER 7

The MERLiN Grammar

The simplest way to communicate with the core of MERLiN is to build a file and send it to MERLiN. However, this file has to obey a specific syntax. This section intends to provide a description of the syntax used to write valid MERLiN files and is broken into two sections:

What Is A Grammar?. This section gives on the use and the meaning of a grammar.

Description of the MERLiN Grammar provides a complete description of the grammar used.

If you feel comfortable with grammar specifications, you can go directly to the section that describes MERLiN's grammar including the list of tokens used in MERLiN and the grammar rules.

What Is A Grammar?

A grammar is a set of *rules* that a file syntax must obey. These rules are a combination of tokens that are compared to tags present in the input file. The comparison (also called parsing) determines whether or not the input file respect entirely the correct syntax. This operation is performed because the application usually expects a certain set of parameters placed in order to build its internal structure with valid values. An example for a rule declaration is as follows:

```
RULE
  TOKEN1
  | TOKEN2
```

This rule is applied in the definition of a component's parameter, MODE_VALUE:

```
MODE_VALUE
  USED
  | NOT_USED
```

This rule means that the mode of a component can be either *USED* or *NOT_USED* (The symbol "|" is a logical symbol used to define different possible values for a rule). Both USED and NOT_USED are tokens, meaning that in the file, we should find the word "USED" or "NOT_USED" at the correct place. A token can also be a rule. Thus rules include a combination of tokens and other rules. For example, the following rule specifies how to declare a link:

```
LINK_DECLARATION_EXPLICIT
  LINK ID ID ID LINK_DEF
LINK_DEF
  OPEN
  DISTANCE ID END
  NB_AMPLIFIER ID END
  NB_REGENERATOR ID END
  FIBER_RATIO ID END
  LAMBDA_RATIO ID END
  BER ID END
  DELAY ID END
  JITTER ID END
  BW ID END
  OP_TYPE OP_TYPE_VALUE END
  MODE MODE_VALUE END
  STATUS STATUS_VALUE END
  CLOSE
```

LINK and ID are tokens but LINK_DEF is a rule. The rule LINK_DEF is defined with tokens and three rules, which are OP_TYPE_VALUE, MODE_VALUE, and STATUS_VALUE.

Each final token can be represented in the file by the word itself or an alias to it. Thus, each token of the grammar can have a substitute that is defined in an alias table. For example, the previous DEFAULT_LINK_DEF rule can be described in the input file as follows:

```
{
     DISTANCE              10 ;
     NB_AMPLIFIER          10 ;
     NB_REGENERATOR        20 ;
     FIBER_RATIO           3 ;
     LAMBDA_RATIO          5 ;
     BER                   0.01 ;
     DELAY                 0.03 ;
     JITTER                0.02 ;
     BW                    155 ;
     OP_TYPE               NORMAL ;
     MODE                  NOT_USED ;
     STATUS                WORKING ;
}
```

Figure 8: Example of the Link's file representation in MERLiN

Here for simplicity, the symbol used to represent the token DISTANCE in the grammar is the word "DISTANCE", but the one used to represent the token END is ";". The tokens must be separated either by a space or by an end of line. The description of the grammar should then contain the list of all the substitutes to the tokens and the list of all the rules.

For more information concerning the construction of grammars and more detailed examples, please refer to the GNU web site of Bison (the GNU parser) at http://www.gnu.org/manual/bison/index.html.

Description of the MERLiN Grammar

This section presents the full description of the MERLiN grammar. The list of tokens is useful in order to know the substitutes that must be present in the script file.

Token list

FIGURE 24. **Tokens' substitutes**

Token	Substitute
ACTION	ACTION
ADD_CONNECTION	ADD_CONNECTION
ALGO_LOCATION	ALGO_LOCATION
ASSIGN_WL	ASSIGN_WL
CLOSE	}
CONNECTION	CONNECTION
CONNECTION_BLOC_TAG	[CONNECTION_BLOC]
CONNECTION_MAX	CONNECTION_MAX
CREATE_ROUTE	CREATE_ROUTE
BER	BER
BW	BW
DEFAULT	DEFAULT
DELAY	DELAY
DEST	DEST
DISTANCE	DISTANCE
END	;
FAILED	FAILED
FIBER_RATIO	FIBER_RATIO
GUI	GUI
ID	["|"":""\\"A-Za-z0-9_/\.-<>]+

FIGURE 24. **Tokens' substitutes**

Token	Substitute
JITTER	JITTER
LAMBDA_LIST	LAMBDA_LIST
LAMBDA_RATIO	LAMBDA_RATIO
LAMBDAS	LAMBDAS
LINK	LINK
LINK_BLOC_TAG	[LINK_BLOC]
LINK_FAILURE	LINK_FAILURE
MEAN_CONNECTION_SET	MEAN_CONNECTION_SET
MODE	MODE
NB_AMPLIFIER	NB_AMPLIFIER
NB_REGENERATOR	NB_REGENERATOR
NODE_BLOC_TAG	[NODE_BLOC]
NODE_FAILURE	NODE_FAILURE
NODE_NAME_LIST	NODE_NAME_LIST
NORMAL	NORMAL
NOT_USED	NOT_USED
OPEN	{
OP_TYPE	OP_TYPE
OXC_CONVERT	OXC_CONVERT
OXC_NON_CONVERT	OXC_NON_CONVERT
PROTECTION	PROTECTION
RECONFIGURATION	RECONFIGURATION
REM_CONNECTION	REM_CONNECTION
ROUTE	ROUTE
ROUTE_BLOC_TAG	[ROUTE_BLOC]
ROUTE_ID	ROUTE_ID
START	START
STATE	STATE

FIGURE 24. Tokens' substitutes

Token	Substitute
STATUS	STATUS
SWITCH_TABLE	SWITCH_TABLE
TOPOLOGY	TOPOLOGY
USED	USED
WAVE	WAVE
WAVE_BLOC_TAG	[WAVE_BLOC]
WL_ASSIGNED_LIST	WL_ASSIGNED_LIST
WORKING	WORKING

Please note that the substitute for the ID token is a regular expression.

Rules list

```
S:
  TOPOLOGY_BLOC ACTION_BLOC GUI_BLOC

TOPOLOGY_BLOC:
  TOPOLOGY OPEN
  NODE_BLOC_TAG OPEN NODE_BLOC_LIST CLOSE
  LINK_BLOC_TAG OPEN LINK_BLOC_LIST CLOSE
  CONNECTION_BLOC_TAG OPEN CONNECTION_BLOC_LIST CLOSE
  ROUTE_BLOC_TAG OPEN ROUTE_BLOC_LIST CLOSE
  WAVE_BLOC_TAG OPEN WAVE_BLOC_LIST CLOSE
  CLOSE

ALGO_LOCATION_DEF:
  ALGO_LOCATION
  ID
  ID
  END

NODE_BLOC_LIST:
  END
  | NODE_BLOC NODE_BLOC_LIST
  | NODE_BLOC
```

```
LINK_BLOC_LIST:
  END
  | LINK_BLOC LINK_BLOC_LIST
  | LINK_BLOC

CONNECTION_BLOC_LIST:
  END
  | CONNECTION_BLOC CONNECTION_BLOC_LIST
  | CONNECTION_BLOC

CONNECTION_SHORT_BLOC_LIST:
  END
  | CONNECTION_SHORT_BLOC CONNECTION_SHORT_BLOC_LIST
  | CONNECTION_SHORT_BLOC

ROUTE_BLOC_LIST:
  END
  | ROUTE_BLOC ROUTE_BLOC_LIST
  | ROUTE_BLOC

WAVE_BLOC_LIST:
  END
  | WAVE_BLOC WAVE_BLOC_LIST
  | WAVE_BLOC

NODE_BLOC:
  DEFAULT_OXC_CONVERT_DECLARATION
  | DEFAULT_OXC_NON_CONVERT_DECLARATION
  | OXC_CONVERT_DECLARATION
  | OXC_NON_CONVERT_DECLARATION

LINK_BLOC:
  DEFAULT_LINK_DECLARATION
  | LINK_DECLARATION

DEFAULT_LINK_DECLARATION:
  DEFAULT LINK LINK_DEF

DEFAULT_OXC_CONVERT_DECLARATION:
  DEFAULT OXC_CONVERT OXC_CONVERT_BLOC
```

```
DEFAULT_OXC_NON_CONVERT_DECLARATION:
  DEFAULT OXC_NON_CONVERT OXC_NON_CONVERT_BLOC

LINK_DECLARATION:
  LINK_DECLARATION_EXPLICIT
  | LINK_DECLARATION_IMPLICIT

LINK_DECLARATION_EXPLICIT:
  LINK ID ID ID LINK_DEF

LINK_DECLARATION_IMPLICIT:
  LINK ID ID ID END

OXC_CONVERT_DECLARATION:
  OXC_CONVERT_DECLARATION_EXPLICIT
  | OXC_CONVERT_DECLARATION_IMPLICIT

OXC_CONVERT_DECLARATION_IMPLICIT:
  OXC_CONVERT ID END

OXC_CONVERT_DECLARATION_EXPLICIT:
  OXC_CONVERT ID OXC_CONVERT_BLOC

OXC_NON_CONVERT_DECLARATION:
  OXC_NON_CONVERT_DECLARATION_EXPLICIT
  | OXC_NON_CONVERT_DECLARATION_IMPLICIT

OXC_NON_CONVERT_DECLARATION_EXPLICIT:
  OXC_NON_CONVERT ID OXC_NON_CONVERT_BLOC

OXC_NON_CONVERT_DECLARATION_IMPLICIT:
  OXC_NON_CONVERT ID END

LINK_DEF:
  OPEN
  DISTANCE ID END
  NB_AMPLIFIER ID END
  NB_REGENERATOR ID END
  FIBER_RATIO ID END
  LAMBDA_RATIO ID END
  BER ID END
  DELAY ID END
  JITTER ID END
  BW ID END
  OP_TYPE OP_TYPE_VALUE END
  MODE MODE_VALUE END
  STATUS STATUS_VALUE END
  LAMBDA_LIST_BLOC
  CLOSE

LAMBDA_LIST_BLOC:
  LAMBDA_LIST OPEN LAMBDA_LIST_DEF CLOSE

LAMBDA_LIST_DEF:
  END
  | LAMBDA_LIST_ELEMENT LAMBDA_LIST_DEF
  | LAMBDA_LIST_ELEMENT

LAMBDA_LIST_ELEMENT:
  ID OP_TYPE_VALUE MODE_VALUE STATUS_VALUE END

OXC_CONVERT_BLOC:
  OPEN
  BER ID END
  DELAY ID END
  JITTER ID END
  BW ID END
  OP_TYPE OP_TYPE_VALUE END
  MODE MODE_VALUE END
  STATUS STATUS_VALUE END
  SWITCH_TABLE_DEF
  CLOSE

OXC_NON_CONVERT_BLOC:
  OPEN
  BER ID END
  DELAY ID END
  JITTER ID END
  BW ID END
  OP_TYPE OP_TYPE_VALUE END
  MODE MODE_VALUE END
  STATUS STATUS_VALUE END
  SWITCH_TABLE_DEF
  CLOSE

SWITCH_TABLE_DEF:
  SWITCH_TABLE OPEN SWITCH_ENTRY_LIST CLOSE

SWITCH_ENTRY_LIST:
  SWITCH_ENTRY SWITCH_ENTRY_LIST
  | SWITCH_ENTRY

SWITCH_ENTRY:
  ID ID ID ID END
  | END

CONNECTION_BLOC:
  CONNECTION ID OPEN
  START ID END
  DEST ID END
  DISTANCE ID END
  BER ID END
  DELAY ID END
  JITTER ID END
  BW ID END
  CLOSE

CONNECTION_SHORT_BLOC:
  CONNECTION ID END

OP_TYPE_VALUE:
  NORMAL
  | PROTECTION

MODE_VALUE:
  USED
  | NOT_USED

STATUS_VALUE:
  WORKING
  | FAILED
```

```
ROUTE_BLOC:
  ROUTE ID OPEN
  CONNECTION ID END
  DISTANCE ID END
  BER ID END
  DELAY ID END
  JITTER ID END
  BW ID END
  NODE_NAME_LIST NODE_NAME_LIST_DEF
  CLOSE

NODE_NAME_LIST_DEF:
  ID NODE_NAME_LIST_DEF
  | END

WAVE_BLOC:
  WAVE OPEN WAVE_DEF CLOSE

WAVE_DEF:
  ROUTE ID END WL_ASSIGNED_LIST
  OPEN WL_ASSIGNED_LIST_DEF CLOSE

WL_ASSIGNED_LIST_DEF:
  WAVE_LAMBDA_LIST_BLOC WL_ASSIGNED_LIST_DEF
  | WAVE_LAMBDA_LIST_BLOC

WAVE_LAMBDA_LIST_BLOC:
  OPEN LINK ID END LAMBDA WAVE_LAMBDA_LIST CLOSE

WAVE_LAMBDA_LIST:
  ID WAVE_LAMBDA_LIST
  | END

ACTION_BLOC:
  ACTION OPEN ACTION_LIST CLOSE

 ACTION_LIST:
  ACTION_DEF ACTION_LIST
  | ACTION_DEF

ACTION_DEF:
  END
  | CREATE_ROUTE_ACTION
  | ASSIGN_WL_ACTION
  | RECONFIGURATION_ACTION
  | ADD_CONNECTION_ACTION
  | REM_CONNECTION_ACTION
  | LINK_FAILURE_ACTION
  | NODE_FAILURE_ACTION

CREATE_ROUTE_ACTION:
  CREATE_ROUTE OPEN ALGO_LOCATION_DEF
  CONNECTION_BLOC_LIST CLOSE

ASSIGN_WL_ACTION:
  ASSIGN_WL OPEN ALGO_LOCATION_DEF
  CONNECTION_SHORT_BLOC_LIST CLOSE

RECONFIGURATION_ACTION:
  RECONFIGURATION
  OPEN
  ALGO_LOCATION_DEF
  CONNECTION_BLOC
  CONNECTION_MAX ID END
  MEAN_CONNECTION_SET ID END
  CLOSE

ADD_CONNECTION_ACTION:
  ADD_CONNECTION ALGO_LOCATION_DEF OPEN CLOSE

REM_CONNECTION_ACTION:
  REM_CONNECTION ALGO_LOCATION_DEF OPEN CLOSE

LINK_FAILURE_ACTION:
  LINK_FAILURE ALGO_LOCATION_DEF OPEN CLOSE

NODE_FAILURE_ACTION:
  NODE_FAILURE ALGO_LOCATION_DEF OPEN CLOSE

GUI_BLOC:
  | GUI OPEN ID_LIST_DEF CLOSE
```

```
ID_LIST_DEF:
  | END
  | ID ID ID END ID_LIST_DEF
```

CHAPTER 8

Extending MERLiN

The communication system included in MERLiN allows the programmers to implement their own algorithms and plug-ins. By thus way, MERLiN can simulate a large variety of algorithms, and can be interfaced with other network simulators such as optical lower layer simulators or higher layer simulators such as NS, NISI, ASM, etc...). The aim of this section is to explain the different structures and modules to use in order to add a new algorithm/plug-in to MERLiN.

The Plug-in section describes how to create a new plug-in such as an interface to another simulator.

The Algorithm section explains how to implement your own algorithm and integrate it to MERLiN database.

The Integration To MERLiN section presents the different steps in the integration of a new module in the MERLiN archive.

The Plug-in

The MERLiN archive includes a simple communication mechanism which takes care of the data exchange between the algorithm/plug-in and the core. A template generic Makefile script is also provided in order to compile your application. In order to implement your own module, you do not have to be familiar with the communication techniques, but it is recommended to have a basic knowledge of the C,C++ or Java languages programming techniques, and compilation techniques.

In order to speed up the development of a plug-in, a communication API has been developed in C++. The code of the API and a template Makefile can be copied from the *src/template/plug-in* directory of the archive. The communication API call a specific function named *plugInMain* that must be present in your module, and will be called with the following signature.

void ***plugInMain*** *(int argc, char **argv);*

The first argument is the number in arguments of the command line, and the second argument represent the command line parameters used in the plug-in. For more details, please refer to the Client class description in the reference manual available online.

The communication API of a plug-in will only take care of opening and closing the communication port (italic letter in the following description). Here is the description in pseudo-code of what the text file plug-in is supposed to do.

- *Connect to MERLiN communication port.*
- *Call the plugInMain function*
- Read an input file.
- Send the file to MERLiN.
- Wait for the reply on the communication port.
- If the answer is an error, return the error to the user.
- Else the answer is an acknowledgement message, read the parameters on the communication port.
- Write the Output file on the disk.
- *return from the plugInMain function*
- *Close the communication port.*

The Algorithm

An algorithm receiving a list of parameters from MERLiN does some computation before returning an output list of parameters to MERLiN.

Similar to the plug-ins, a communication API for the algorithm has been developed in C++. The code of the API and a template Makefile can be copied from the *src/template/algorithm* directory of the archive. The communication API starts the communication between the algorithm and the core and get the parameters needed. Then it calls a specific function named *algoMain*. This function MUST be present in your module, and will be called with the following signature.

void ***algoMain*** *(ParameterList & param_in_list,*
ParameterList & param_out_list);

The arguments are respectivly the list of parameters needed as an input for this algorithm (filled by the communication API), and the output list of parameters for this algorithm (this must be filled by the algorithm). The output list of parameters contains the modified topology and associated components, and the performance parameters of the algorithm. The modified list of parameters must be retrieved by the communication API in order to end properly the communication process with MERLiN. You are free to do anything necessary in your algorithm but you won't be able to communicate with the core until the algorithm completes its execution. Beside these constraints you are allowed to implement whatever you want in your algorithm.

We list here the pseudo-code of an algorithm used in MERLiN. The code attempts to give an idea of the mechanisms used for communicating between the different elements of the simulator. The communication API (italic in the following description) is a little bit more complex than for a Plug-In.

- *Open a communication port.*
- *Register the algorithm to MERLiN.*
- *Wait for a command from MERLiN.*
- *Read the message from the communication port.*
- *If the command is invalid, return an error to MERLiN.*
- *If the command is valid, read the parameters on the communication port.*
- Build the internal structures (topology, route, set algorithms parameters)

- Execute the algoMain function of the algorithm.
- *Return the results to MERLiN.*
- *Close the communication port.*

To identify and register your algorithm in MERLiN's core, you MUST also provide in your module a reference to a special string that is the name of the executable file, result of the compilation process of your algorithm. Please declare at the begining of your module the following variable:

*char * algorithm_id = "[path/]name_of_the_algorithm";*

This string MUST NOT contain any special characters that could interfere with the file system of your computer. It must contain either the full path from the root directory to the binary file, or the relative path from the directory defined in the variable MERLINPATH. We recommend the second solution. The path should be set to *algorithm/algorithm_class* to match what has already be implemented in the current version of the simulator.

Integration To MERLiN

We first recommend to place the source files of your new module in a specific directory, and place it into the corresponding sub-directory of the source tree. See the description of the sub-directories inside the *src* directory of the archive (Part1, Chapter). Every time that you want to implement an algorithm or a plug-in, you should take a look at the template Makefile and modify the variables located in the part that is outside the WARNING statements. You will have eventually to modify the Object compilation part (at the end of the template Makefile) in order to compile your own modules.

We recommend the use of the template makefile in order to integrate more easily your own modules to MERLiN. This file is named *Makefile.in* and is not (and should not be) a valid makefile that can be used directly by anyone. After a small modification in the **configure** script of the archive, the compilation process will take care of the modifications inside the template. The first step in the integration of your module to MERLiN is to make the modifications inside the *Makefile.in* file as described in the following example (used for the shortest path algorithm).

```
EXE= shortest_path
```

```
INSTALL_EXE= algorithm/routing
OBJECTS=
…
```

Please provide a type for the INSTALL_EXE variable. This will allow MERLiN to install your executable code in the corresponding categories of algorithms. The type can be:

- routing(Routing Algorithms)
- wavelength(Wavelength Assignment Algorithms)
- reconfiguration(Reconfiguration Algorithms)
- qos(Quality Of Service Algorithms)

You can then add your own list of modules to the compilation process, by setting the OBJECTS variable. Then you will have to add the compilation lines for each module in the Object Section of the template Makefile.

To generate the Makefile associated to the template you have to run the **configure** script in order to fill in the directories and the compilation variables that match with your architecture. Then you can go to MERLiN root directory and launch the configure script with the --file option on the file that you want to generate (here the Makefile):

Example:./configure --file=path_to_my_algorithm/Makefile

Then you can go back to your working directory and compile your algorithm. If there are no errors you can follow the next step in order to integrate your algorithm in MERLiN.

Here is a table representing the differents actions that one can perform and ask to MERLiN.

IN	TOPOLOGY	List <Component>
	CREATE_ROUTE	List <Connection>
OUT	TOPOLOGY	List <Component> (modified)
	ROUTE_CREATED	List <Route>
IN	TOPOLOGY	List <Component>
	ASSIGN_WAVELENGTH	List <Route>

OUT	TOPOLOGY	List <Component> (modified)
	WAVELENGTH_ASSIGNED	List <Wave>
IN	TOPOLOGY	List <Component>
	ADD_CONNECTION	List <Connection>
		List <Route>
OUT	TOPOLOGY	List <Component> (modified)
	CONNECTION_ADDED	List <Route> (new)
		List<Route> (modified)
IN	TOPOLOGY	List <Component>
	REMOVE_CONNECTION	List <Connection>
		List <Route>
OUT	TOPOLOGY	List <Component> (modified)
	CONNECTION_REMOVED	List <Route> (removed)
		List <Route> (modified)
IN	TOPOLOGY	List <Component>
	FAIL_LINK	List <Link>
		List <Wave>
		List <Routes>
OUT	TOPOLOGY	List <Component> (modified)
	LINK_FAILED	List <Wave>
		List <Route>
		List <Route> (modified)
IN	TOPOLOGY	List <Component>
	FAIL_NODE	List <Node>
		List <Wave>
		List <Route>
OUT	TOPOLOGY	List <Component>

	NODE_FAILED	List <Wave> List <Route> List <Wave> (modified) List <Route> (modified)
IN	TOPOLOGY FAIL_LAMBDA	List <Component> List <Lambda> List <Link> List <Wave> List <Route>
OUT	TOPOLOGY LAMBDA_FAILED	List <Component> List <Link> List <Wave> List <Route> List <Link> (modified) List <Wave> (modified) List <Route> (modified)
IN	TOPOLOGY RECONFIGURATION	List<Component> List<Connection> NbMaxConnection MeanNbConnection
OUT	TOPOLOGY RECONFIGURED	List<Component> (modified) List<Route> List<Wave>

Appendices

CHAPTER 9

k shortest path

Abstract

The k_shortest_path is a routing algorithm that computes K shortest paths for a connection request between a node pair. For now, K is fixed to 3, but in the next version of MERLiN the user will be able to set this number.

Input parameters

Input parameters are :

a description of the current topology that contains :

- a list of links,
- a list of nodes (cross connect),
- a list of existing routes, that the algorithm has to consider in computing new routes,
- a list of connection requests.

Output parameters

- a list of new computed routes.
- a list of all routes currently set in the topology. This includes the previously existing routes in the given topology and the new computed routes.

Pseudo code

We try to find the K shortest routes between *source_node* and *destination_node*. The main loop of the algorithm is recursive. The structure is as follows :

```
begin

1. Define a set of possible routes P and a set of solutions S.
Initially P = {source_node}, S = {}
2. Look for all available links from each element of P and construct a temporary
   set of paths T.

 for each route Ri in P do
   for each node j neighbor of the last node of Ri do
     tmpR = Ri + j (add the node j after the last node of Ri)
     T = T + tmpR
   end do
 end do

3. Select a minimum cost path from the temporary set T
     selected_path = get_path_of_minimum_cost_from (T)
     P = P + selected_path
4. If selected_path is a solution (i.e. the last node of selected_path is
   destination_node), then S = S + selected_path.
5. Go to 2 until S has K elements.

end
```

Structures and temporary formats

The algorithm uses a class called KRoute. Each element of the set P is an instance of KRoute. The main characteristics of a KRoute are :

- ml_path:
  the path used by KRoute.
- mi_cost:
  the cost of ml_path. Each time a node is added to the path, the cost is recomputed.
- ml_try:
  a list of possible paths constructed as follows: during phase 2 of the algorithm (see above), each neighbor node of the last node of ml_path is added to ml_path. The new resulting path is put in ml_try.
- mp_best_kroute
  this is the element of ml_try that has the lowest cost.

A KRoute is a possible route for a given connection and is placed in the P set.

Integration in MERLiN

Path:	$MERLINPATH/algorithm/routing/k_shortest_path
Category:	routing
Restrictions:	none

References

The k-shortest path is a common algorithm which has been studied intensivly. Almost every paper on the k-shortest path describes a basic form of this algorithm. A good starting point to find informations on operations research algorithms is : http://mat.gsia.cmu.edu.

CHAPTER 10

Tabu

Abstract

The tabu algorithm is a wavelength allocation algorithm based on the tabu search paradigm. This algorithm trie to allocate one wavelength for each route. Wavelength conversion is not considered along a route.

Input parameters

a description of the current topology that includes :

- a list of links
- a list of nodes (cross connects)
- a list of existing routes, that the algorithm has to consider in computing new routes.

Output parameters

- a list of new computed routes.
- a list of all routes currently set on the topology.

Pseudo code

The number of neighbors the tabu algorithm look at is called *neighbor_size*, and is the size of the *solution table.*

```
begin
Create a graph derived from the input topology. (This graph is a set of linked vertices. See the part "Structures and temporary formats")
6. Make a random coloring on this graph.
7. Compute the conflicts existing and fill the conflict list whith doubles (vertex, number of conflicts)

iteration = 0
while ( size of conflict list != 0  and iteration < 1000 )
      clear the solutions table
      for neighbor_size time do
         2a. p_vertex = pick a vertex in the graph
         2b. p_color  = pick a color
         if ( ( p_vertex.current_color != p_color) AND
              ( p_vertex, p_color is not in the tabu list ) AND
              ( p_color is available on p_vertex) )
         then
            compute the number of conflicts resulting of the potential allocation of p_color to p_vertex
            create a solution wich is the tuple ( array of the color used by each vertex, nb_conflicts )
            put this solution in the solution table
         else goto 2a
      end for
      take the best solution from the solution table
      put the best solution in the tabu list
```

```
        update the *conflict list*
        iteration++
**end while**
**end**
```

Structures and temporary formats

In order to allocate a wavelength, we use a graph derived from the input network. In this graph the nodes are called vertex, and each vertex is associated with a route. There is a link between two vertices if the routes they are associated with have at least one common link. Allocating a wavelength for the routes of the input network is equivalent to color the derived graph. The tabu paradigm is used to color the derived graph.

The algorithm use three structures:

1. The vertex

The attributes of a vertex are :

- mp_route
  the route it is associated with.
- mi_color
  The wavelength allocated to mp_route.
- ml_link
  a list of the links of the route. This is used to achieve two objectives :
  1 know what vertices are neighbors, i.e. they have a link in common.
  2 know what wavelengths are available along the links of the route.
- ml_neighbor
  a list of the vertex's neighbors. It is used to know if the coloring of the vertex generates a conflict with its neighbors.
- mb_is_already_allocated
  a boolean indicating that this route has already been allocated, so its color need not to be changed.

2. The tabusol

A tabusol represents a move in the tabu algorithm. A tabusol is an entry in the tabu list. This is basically one of the best choices available, i.e. a choice that generates the least amount of conflicts.

The attributes of a tabusol are :

- topo
  an Array which size is equal to the number of routes. Each element of it is a color associated to a route.

3. The sol

A sol represents a possible change of color for a vertex.
The attributes of a sol are :

- vertex
  the vertex whose color can be changed.
- color
  the new color.
- conflicts
  number of conflicts resulting from changing the vertex’color to the new color.
- ml_vertex_to_add
  list of vertices that should be added to the conflict list if the color of the vertex is changed.
- ml_vertex_to_remove
  list of vertices that should be removed from the conflict list if the color of the vertex is changed.

The algorithm use three main lists ;

1. the conflicts list
   stores all vertices having the same color as their neighbor’s color.
2. the tabu list
   stores every tabu "move", i.e. every best choice choosen after a given number of iterations.
3. the solution table
   this is a temporary table where the best solution is picked and put into the tabu list.

The size of the tabu list and the solution table is fixed to 10, but it can be set as variable parameter in order to adapt the algorithm to a given topology and traffic matrix.

Integration in MERLiN

Path: $MERLINPATH/algorithm/wavelength/tabu
Category: wavelength
Restrictions: this algorithm does a wavelength allocation without wavelength conversion.

References

For an overview of the tabu search paradigm, see http://www.winforms.phil.tu-bs.de/winforms/research/tabu/tabu.html

CHAPTER 11

First Fit Reconfiguration

Abstract

The first_fit_reconf algorithm is an example of reconfiguration algorithm. It allocates wavelengths to a set of connection requests generated overtime according to a normal distribution. It uses the functions used by the shortest_path for the routing, and the functions used by the first_fit for the wavelength allocation.

Input parameters

The input parameters are :

a description of the current topology containing :

- a list of links
- a list of nodes (cross connect)
- the mean value of the request arrivals
- the maximum number of connection requests generated

Output parameters

- a list of the new computed routes.
- a list of all routes in the current topology. It includes the just computed routes.

Pseudo code

```
begin
while (number_of_generated_connection < limit_number_of_generated-connection)
    set_of_connection = generate a set of connection according to a normal distribution
    number_of_generated_connection += size ( set_of_connection )
    shortest_path ( set_of_connection )
    first_fit ( set_of_connection )
end while
end
```

Structures and temporary formats

There are no structures specific to this algorithm.

Integration in MERLiN

Path:	$MERLINPATH/algorithm/reconfiguration/first_fit_reconf
Category:	reconfiguration
Restrictions:	no restriction